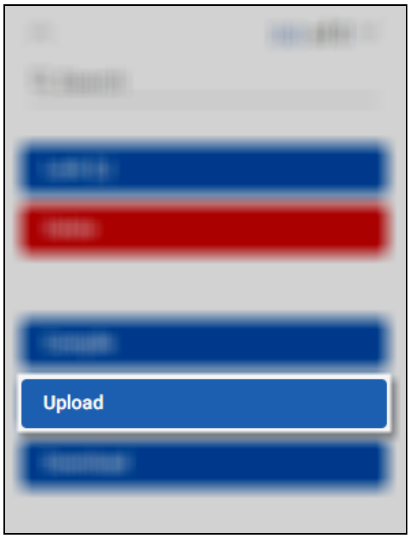
smarsh®

Cognition Studio

User Guide

August 28, 2024

Table of Contents



Next, select a .txt file with your predifined terms. Otherwise, you must individually add terms to your Lexicon.

Note

Any uploaded .txt files should contain the raw text of your Lexicon terms separated by line breaks. Files are limited in size to 300kb. The terms in the file should conform to the selected Natural Language or Regex grammar (Natural Language, Regex) or else they will fail compilation. You can download a template .txt file and poopulate it with your own terms using the following link: Template Examples for Lexicon Term Uploads.txt.

Lexicon Compile Modes

The Compile Mode concept is key to understanding how Cognition Studio interprets and applies Lexicons during runtime.

What is a Compile Mode?

A **Compile Mode** transforms Lexicon terms into a format to match text during runtime. This process includes normalization and preprocessing steps packaged into a Portable Message Format (PMF) archive.

Jump ahead to the available Compile Modes using the following links:

During runtime, the PMF archive applies normalization to incoming text and matches it against Lexicon terms. Normalization can change the length of the text, so matching spans have to be adjusted to align with the original text. Cognition Studio Compile Modes perform this change automatically.

A Compile Mode includes several components:

- **Grammar**: Determines valid terms and their interpretation. See Grammars below for more information. Universal Grammar is the default Compile Mode grammar.
- **Translation**: Converts Lexicon terms into an intermediate form for use by a Compiler/Backend pair.
- **Backends (compiler pairs)**: Compile the intermediate form into something that the backend can execute.
- **Configuration**: Settings for translation and compilation operations.
- **Normalization/Preprocessing**: Process applied to text before matching against Lexicon terms.

Note

Cognition Lexicons are **case insensitive**. This means that a Lexicon term like "Hello" will produce matches on the following versions of the target text:

- "HELLO"
- "HeLo"
- "hello"

This applies to Universal, Natural Language, and Regex Compile Modes.

Note

Preprocessing is **not** applied to character-based Compile Modes.

Note

The definition of a Compile Mode sets all configurations. You can't make changes after the definition to ensure consistent behavior for the Lexicons using that Compile Mode.

The following pages include additional information about Lexicons and Lexicon Compile Modes, including best practices and how Cognition processes emojis and emoticons:

- Changing the Compile Mode
- Compile Mode best practices
- Lexicon Emoji and Emoticon Behavior

Grammars

Grammars determine the Lexicon syntax, meaning which terms are valid and how they're interpreted during processing. Cognition Studio supports **Universal Lexicon**, **Natural Language**, and **Regex** grammars.

Universal Lexicon Grammar is a single Compile Mode that handles both Natural Language and regex expressions. You can combine Natural Language and regex in a single Lexicon term with Universal Grammar. Universal Grammar also offers exclusive, expanded support for Unicode character classes and Unicode punctuation normalization during Lexicon runtime. Universal Grammar is the **default** Compile Mode in Cognition Studio.

Note

When using a multi-character wildcard (`*`) in a Universal Grammar Lexicon term, your ASCII-based expressions must contain at least three (3) characters to be valid. For example:

- `h*ey` is valid because there are three (3) characters in the term other than the wildcard (`*`).
- `h*e*` is not valid because there are only two (2) characters in the expression that aren't wildcards (`*` , `*`).
- `h*ey h*e*` is not valid because the second expression in that term is invalid just like the above example.
- `h*ey FOLLOWEDBY{2} h*e*` is also invalid.
- However, `漢*` is valid with only a single character other than the wildcard. This is because the three character rule applies to ASCII characters. Non-ASCII expressions invalidate this rule.

Natural Language is a human-readable Lexicon definition language. It supports connecting words or phrases together with Boolean operators (`AND` , `OR` , `NOT`), proximity operators (`NEAR{n}` , `FOLLOWEDBY{n})` , single-character wildcards (`?`), and multi-character wildcards (`*`).

Note

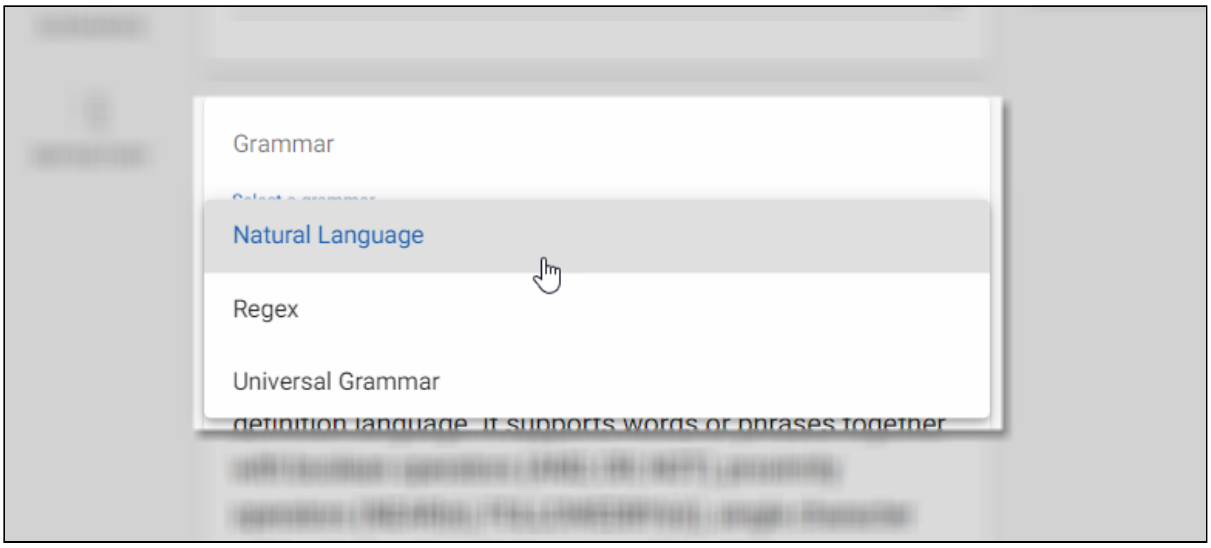
Natural Language grammar collapses multiple, consecutive whitespace characters between two character tokens into one whitespace character.

Regex (also known as regular expressions) is a string of text describing a search pattern. These expressions can contain wildcards, quantifiers, groups, and Boolean `OR` operators. Regex is frequently used as a machine-readable search definition. It can be produced by other systems and used as Lexicon term definitions. For example: `trad.*` or `\b[m]\w+` are valid expressions.

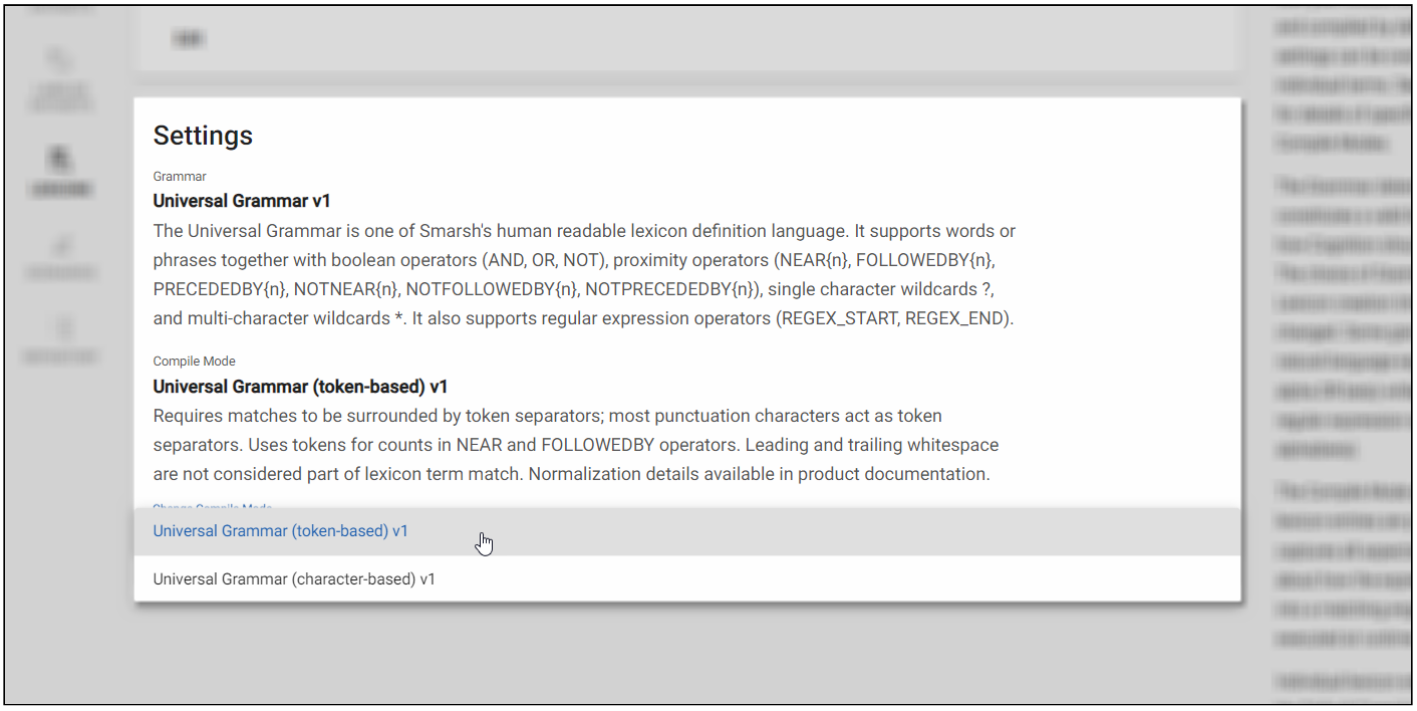
Note

Regex grammar requires Lexicon creators and users to determine how multiple consecutive whitespace characters are handled. You can use regex in Lexicon terms to make that determination.

Grammar types are selected when creating a new Lexicon. The Universal Grammar is the default option in Cognition Studio. To use a regex or Natural Language grammar, you must manually select it.



After selecting a grammar type, you then select the explicit Compile Mode associated with the chosen grammar from the **Settings** menu of the Lexicon itself.



Note

Your choice of grammar is fixed at the creation of a new Lexicon and can't be changed.

Universal Lexicon Grammar

Follows the expressions defined within Natural Language grammars. It adds the following additional operators:

- **REGEX_START**
- **REGEX_END**
- **NOTNEAR**
- **NOTFOLLOWEDBY**
- **PRECEDEDBY**
- **NOTPRECEDEDBY**

The operators `REGEX_START` and `REGEX_END` tell Cognition to treat the text within these operators as regex. Text outside of these distinct operators is treated the same as Natural Language operators.

Note

The Universal Grammar doesn't accept Lexicon terms matching single ASCII character expressions. This includes Natural Language terms (`a`) and regex terms (`REGEX_START(a)REGEX_END`). It also applies to other operators, such as parentheses (`(a)`) and negations (`NOT a`).

Preprocessing

Universal Grammar also offers exclusive, expanded support for Unicode character classes and Unicode punctuation preprocessing during Lexicon runtime. The following table details this additional Unicode support:

Definition	Character Categories	Notes
Alphanumeric	Click the links for a full list of each category: • Lu - Letter, Uppercase • Ll - Letter, Lowercase • Lt - Letter, Titlecase • Lm - Letter, Modifier • Lo - Letter, Other • Nl - Number, letter • Nd - Numbera, decimal	This category includes non-English language alphanumeric characters as well, like `dové` or `क्षेत्र ५१`.
Punctuation	Click the links for a full list of each category: • Pc - Punctuation, connector • Pd - Punctuation, dash • Ps - Punctuation, open • Pe - Punctuation, close • Pi - Punctuation, initial quote • Pf - Punctuation, final quote • Po - Punctuation, other	• This category includes non-English language punctuation as well, like `¿.` • `$~+=<>|^\`` aren't considered punctuation characters.
Dashes	• Pd - Punctuation, dash	This category includes non-English language dash characters, like `–` or `≠`.
Apostrophes	• `'` (U+0027 Apostrophe) • `’` (U+2019 Right single quotation mark [RSQM])	

Definition	Character Categories	Notes
Symbols	• `~` (U+007E Tilde) • `+` (U+002B Plus) • `=` (U+003D Equals) • `<` (U+003C Less than) • `>` (U+003E Greater than) • `|` (U+007C Vertical line) • `^` (U+005E Circumflex accent) • `` ` `` (U+0060 Grave accent)	

Compile Mode Support

Both Universal Grammar Compile Modes, **Universal Grammar Token-Based v1** and **Universal Grammar Character-Based v1**, include enhanced normalization rules compared to Natural Language and regex modes. The following table summarizes these enhancements:

Compile Mode	Normalization Enhancements
Universal Grammar Token-Based v1	• **Unicode Characters class support** • The intermediate regex format for each start with `(?U)` flag instead of `(?u)` that enables the Unicode character class and Unicode-aware case folding. This ensures a regex like `\p{Digit}` matches on non-English language numerals such as `१` along with English numerals such as `[0-9]` . • All Punctuation (except **Dashes** and **Apostrophes**) and **Symbols** are converted to a whitespace when surrounded by **Alphanumeric** characters. • All Punctuation (except **Dashes**) and **Symbols** are converted to a whitespace when preceded by an **Alphanumeric** character. • All Punctuation (except **Dashes**) and **Symbols** are converted to a whitespace when succeeded by an **Alphanumeric** character.

Compile Mode	Normalization Enhancements
Universal Grammar Character-Based v1	• **Unicode Characters class support** • The intermediate regex format for each start with `(?U)` flag instead of `(?u)` that enables the Unicode character class and Unicode-aware case folding.

Natural Language

This grammar is designed to match text either on token-based or character-based boundaries depending on the Compile Mode selected.

Generally, this grammar:

- Accepts characters from all Unicode General Categories.
- Bases the window size for `{n}` in `NEAR{n}` and `FOLLOWEDBY{n}` expressions on the token counts for token-based Compile Modes or character counts for character-based Compile Modes.
- Can combine expression types to construct more complex expressions. For example: `(trade* NEAR{3} share*) AND ((call OR phone OR contact) NEAR{2} me)`.
- Uses a token-based Compile Mode to match on token boundaries. For example, if a Lexicon term is "`alpha`" then it will not match "`alphas`." However, character-based Compile Modes do match on character boundaries. If a Lexicon term is "`alpha`" then it would match "`alphas`."

This grammar accepts the following expressions:

Expression	Definition	Examples

Atomic	An atomic expression has no logical operators.
	A **simple** atomic expression has no wildcards. It makes exact matches between itself and the text.
	A **wildcard** atomic expression contains one, or both, of the `?` (question mark) and `*` (asterisk) symbols.
	• The `?` (question mark) wildcard represents a numerical character range of **1**. It indicates whether any single, non-whitespace character matches.
	• The `*` (asterisk) wildcard represents **0 to unlimited** characters. It indicates a contiguous sequence of non-whitespace character matches.
		• The **wildcard** expression "he*o" matches both:
		  • ✅ "They said hello"
		  • ✅ "The helo returned to base."
		• The **wildcard** expression "he?o" matches both:
		  • ✅ "The helo returned to base"
		  • ✅ "The hero returned to base."
		• The **simple** expression "hello":
		  • ✅ Matches "They said hello"
		  • ❌ Doesn't match "The helo returned to base."
		• The **wildcard** expression "he?o":
		  • ✅ Matches "The hero saved the day"
		  • ❌ Doesn't match "She said hello to him."

Expression	Definition	Examples
OR	The expression " `<exp1> OR <exp2>` " matches text whenever either `<exp>`, `<exp2>`, or both match the text.	• ✅ The expression "alpha OR beta" matches "alpha romeo." • ✅ The expression "alpha OR beta" matches "beta testers." • ❌ The expression "alpha OR beta" doesn't match "gamma ray."

AND	The expression " `<exp1> AND <exp2>` " matches text whenever both `<exp1>` **and** `<exp2>` match anywhere in the text.	• ✅ The expression "alpha AND beta" matches "alpha romeo for beta testers."
		• ✅ The expression "alpha AND beta" matches "beta testers like an alpha romeo."
		• The expression "alpha AND beta" doesn't match:
		  ◦ ❌ "alpha romeo"
		  ◦ ❌ "beta testers."

NEAR{n}	The expression " `<exp1> NEAR{n} <exp2>` " when `<exp1>` and `<exp2>` matches with no more than `N` tokens between them. This is a more constrained form of conjunction. **Note** Only the content between `<exp1>` and `<exp2>` counts against the `{n}` token window.	• ✅ The expression "alpha NEAR{2} beta" matches "alpha romeo for beta testers" • ✅ The expression "alpha NEAR{2} beta" matches "beta testers for alpha romeo" • ❌ The expression "alpha NEAR{2} beta" doesn't match "beta testers like an alpha romeo" • ❌ The expression "alpha NEAR{2} beta" doesn't match "beta testers like gamma rays"
FOLLOWEDBY{n}	The expression " `<exp1> FOLLOWEDBY{n} <exp2>` " matches text when `<exp2>` matches after `<exp1>` with no more than `N` tokens between them. This is a more constrained form of conjunctive and near expressions. **Note** Only the content between `<exp1>` and `<exp2>` counts against the `{n}` token window.	• ✅ The expression "alpha FOLLOWEDBY{2} beta" matches "alpha romeo for beta testers" • ❌ The expression "alpha FOLLOWEDBY{2} beta" doesn't match "beta testers for alpha romeo" • ❌ The expression "alpha FOLLOWEDBY{2} beta" doesn't match "alpha romeo for good beta testers"

Expression	Definition	Examples

NOT <exp>	The `NOT` operator is used for matching against the absence of expression (`<exp#>`) in the text.	• Non-matching examples:
		  • ❌ "NOT(james bond)"
		  • ❌ "apple NOT NEAR{10} banana"
		  • ❌ "apple AND NOT NEAR{10} banana"
		• Matching examples:
		  • ✅ "bond AND (NOT (james bond))"
		  • ✅ "apple AND (NOT (apple NEAR{10} banana))"

Note

You must escape left bracket ([) and right bracket (]) characters when using Natural Language compile modes. These are regex-reserved characters that aren't automatically escaped during compilation. See Known Issues for more information.

Regex

This grammar accepts regular expressions as defined by the Java regular expression syntax. In the regex grammar, Lexicon terms are simply regular expressions. This is particularly useful for uploading a pre-existing collection of regular expressions.

Natural Language and Regex Preprocessing

Preprocessing is applied to text before matching against your Lexicon terms. The core Compile Modes available in Cognition Studio follow different preprocessing rules for alphanumeric and punctuation character classes.

Alphanumeric sequences are the set of characters encapsulating **[A-Za-z0-9]**. Examples include:

- abcde
- abc123

Non-alphanumeric sequences include:

- abcdé

Punctuation classes are the set of characters encapsulating **!"#%&'()*+,./:;<=>?@[\]^_`{|}~$-** as defined under the **POSIX \p{Punct}** class of ASCII characters.

Note

The following Non-ASCII quotation mark characters aren't included in the POSIX \p{Punct} characters. They aren't processed or normalized:

- Single left (‘ [U+2018])
- Single right (’ [U+2019])
- Double left (“ [U+201C])
- Double right (” [U+201D])

Generally, preprocessing:

- Removes the byte order mark (BOM) character.
- Doesn't processing any leading and trailing whitespace surrounding Lexicon terms.
- Normalizes whitespace characters.
- Normalizes punctuation.

Note

Preprocessing is **not** applied to character-based Compile Modes.

The following table describes how whitespace and punctuation normalization occurs:

Preprocessing	Characters	Example
Whitespace normalization occurs for the characters **!"#%&'()*+,./:;<=>?@[\]^_`{|}~** when **preceding** alphanumeric text.	**!"#%&'()*+,./:;<=>?@[\]^_`{|}~**	`'devil` is transformed to `<SPACE>devil`.
Whitespace normalization occurs for the characters **!"#%&'()*+,./:;<=>?@[\]^_`{|}~** when **succeeding** alphanumeric text.	**!"#%&'()*+,./:;<=>?@[\]^_`{|}~**	`devil'` is transformed to `devil<SPACE>`.
Whitespace normalization occurs for the characters **!"#%&'()*+,./:;<=>?@[\]^_`{|}~** when **surrounded by** alphanumeric text.	**!"#%&'()*+,./:;<=>?@[\]^_`{|}~**	`123#devil` is transformed to `123<SPACE>devil`.
Whitespace normalization **doesn't** occur for the characters **$- (minus, hypen)** when **preceding** alphanumeric text.	**$** and **- (minus, hyphen)**	`$devil123` would remain `$devil123` after normalization.
Whitespace normalization **doesn't** occur for the characters **$- (minus, hypen)** when **succeeding** alphanumeric text.	**$** and **- (minus, hyphen)**	`devil123-` would remain `devil123-` after normalization.
Whitespace normalization **doesn't** occur for the characters **' (apostrophe)**, **$**, and **- (minus, hyphen)** when **surrounded by** alphanumeric text.	**' (apostrophe)**, **$**, and **- (minus, hyphen)**	`devil'123` would remain `devil'123` after normalization.

Compile Modes

The Compile Modes can be categorized into two groups based on whether it generates matches conforming to token boundaries or character boundaries.

Universal Grammar Token-Based v1

Default Compile Mode	Yes
Use	Used to combine operator functionality of both Natural Language and regex grammars.
Limitations	This Compile Mode doesn't match on most punctuation characters; instead, they act as token separators. It uses token counts to calculate the window size `{ n}` expressions.
Preprocessing	Follows Universal Lexicon Grammar preprocessing rules, with additional Unicode pattern support, outlined in Preprocessing.
Grammar	Universal Grammar

The Universal Grammar Token-Based Compile Mode comprises the Universal Grammar as its source. It uses a specific set of preprocessing (normalization) rules that are applied to ingested text before generated regex is matched. This mode normalizes Unicode whitespace. Previously, Unicode whitespace was treated differently than ASCII whitespace and matched literally. Unicode whitespace is properly matched with type of whitespace any number of times.

Universal Grammar contains new operators unique to the grammar. While Token and Character-based methods follow the Natural Language design, the following table details the exclusive operators only available when using Universal Grammar Token or Character-Based.

Expression	Definition	Examples
NOTFOLLOWEDBY{n}	The expression " `<exp1> NOTFOLLOWEDBY{n} <exp2>` " matches text when " `<exp1>` " matches and has more than `N` tokens after it and " `<exp2>` ". It produces a valid match if " `<exp1>` " is present but " `<exp2>` " isn't.	• ✅ The expression "alpha NOTFOLLOWEDBY{2} beta" matches "alpha romeo for good beta testers." • ✅ The expression "alpha NOTFOLLOWEDBY{2} beta" matches "testers for the alpha romeo." • ❌ The expression "alpha NOTFOLLOWEDBY{2} beta" doesn't match "alpha romeo for beta testers." • ❌ The expression "alpha NOTFOLLOWEDBY{2} beta" doesn't match "beta testers like gamma rays."

Expression	Definition	Examples
NOTNEAR{n}	The expression "`<exp1> NOTNEAR{N} <exp2>`" matches text when "`<exp1>`" matches and has more than `N` tokens before or after and "`<exp2>`". It produces a valid match if "`<exp1>`" is present but "`<exp2>`" isn't.	✅ The expression “alpha NOTNEAR{2} beta” matches “alpha romeo for good beta testers.” ✅ The expression “alpha NOTNEAR{2} beta” matches “beta testers for the alpha romeo.” ✅ The expression “alpha NOTNEAR{2} beta” matches "testers for the alpha romeo.” ❌ The expression “alpha NOTNEAR{2} beta” doesn't match “alpha romeo beta testers.” ❌ The expression “alpha NOTNEAR{2} beta” doesn't match “beta for alpha romeo.” ❌ The expression “alpha NOTNEAR{2} beta” doesn't match “beta testers like gamma rays.”
PRECEDEDBY{n}	The expression "`<exp1> PRECEDEDBY{n} <exp2>`" matches text when "`<exp2>`" matches before "`<exp1>`" with no more than `N` tokens between them.	✅ The expression “alpha PRECEDEDBY{2} beta” matches “beta testers for alpha romeo.” ❌ The expression “alpha PRECEDEDBY{2} beta” doesn't match “beta testers like an alpha romeo.” ❌ The expression “alpha PRECEDEDBY{2} beta” doesn't match “alpha romeo for beta testers.”
NOTPRECEDEDBY{n}	The expression "`<exp1> NOTPRECEDEDBY{n} <exp2>`" matches text when "`<exp1>`" matches and has more than `N` tokens before it and "`<exp2>`". It produces a valid match if `<exp1>` is present but `<exp2>` isn't.	✅ The expression “alpha NOTPRECEDEDBY{2} beta” matches “beta testers like an alpha romeo.” ✅ The expression “alpha NOTPRECEDEDBY{2} beta” matches "testers for the alpha romeo.” ❌ The expression “alpha NOTPRECEDEDBY{2} beta” doesn't match “beta testers for alpha romeo.” ❌ The expression “alpha NOTFOLLOWEDBY{2} beta” doesn't match “beta testers like gamma rays.”

Expression	Definition	Examples
REGEX_START(and)REGEX_END	The "`REGEX_START(` and `)REGEX_END`" operators, when used together, signify the beginning and end of a Regular Expression pattern. Use this operator to treat the matched text as literally regex content. It matches when the expression between the operators is matched in the text.	• ✅ The expression "REGEX_START(\d{2}/\d{2}/\d{4})REGEX_END" matches "01/01/2024." • ❌ The expression "REGEX_START(\d{2}/\d{2}/\d{4})REGEX_END" doesn't match "1/1/24."

Universal Grammar Character-Based v1

Default Compile Mode	Yes
Use	Used to combine operator functionality of both Natural Language and regex grammars.
Limitations	This Compile Mode doesn't apply any preprocessing. It uses character counts to calculate the window size `n` in `NEAR{n}` related `{n}` expressions. ❗ Using `*` with this Compile Mode could result in an indefinite number of matches on long text.
Preprocessing	**Important** No preprocessing is applied when using this Compile Mode.
Grammar	Universal Grammar

As previously mentioned, Universal Grammar follows the same rules as Natural Language and regex operators. The previous examples listed above of Lexicon terms are applicable to this grammar. Note that you must wrap text you want matched as regex with `REGEX_START` and `REGEX_END`. Terms compiled with the Universal Grammar Character-Based Compile Mode don't enforce a space requirement between a Natural Language expression and a regex expression for a Lexicon term. For example, previously the term `hello REGEX_START(world)REGEX_END` would be compiled as `(hello\s+world)`. Instead, this Compile Mode compiles it as `(helloworld)`.

Universal Grammar contains new operators unique to the grammar. While Token and Character-based methods follow the Natural Language design, the following table details the exclusive operators only available when using Universal Grammar Token or Character-Based.

Lexicon Terms	Text to Match	Match Count	Notes
AB NOTFOLLOWEDBY{3} CD	ABUVWXCDYZ	1	There are more than three (3) characters between AB and CD.
AB NOTNEAR{3} CD	TRMVWQABYZ	1	AB exists and CD doesn't, which produces a match because CD isn't "near."
CD PRECEDEDBY{3} AB	AB12CD	1	
CD NOTPRECEDEDBY{2} AB	AB123CD	1	AB is more than two (2) characters away from CD.
call FOLLOWEDBY{2} REGEX_START(\(?\d{3}\)?[\s.-]\d{3}[\s.-]\d{4}$)REGEX_END	call me at 123-456-7890	1	The provided regex matches on phone number patterns. It matches the hyphen because this Compile Mode is character-based.

Natural Language Token-Based v3

Default Compile Mode	No
Use	Used for Lexicons supporting Natural Language terms where the matching text uses whitespace to separate tokens.
Limitations	This Compile Mode doesn't match on most punctuation characters; instead, they act as token separators. It uses token counts to calculate the window size `n` in `NEAR{n}` and `FOLLOWEDBY{n}` expressions.
Preprocessing	Follows the general preprocessing rules outlined in Preprocessing.
Grammar	Natural Language

The following table includes matches using this Compile Mode:

Lexicon Term	Generated Regex	Text to Match	Normalized Version of Text to Match	Match Count	Notes
Bonjour	(^|\s)(Bonjour)\s	French, Bonjour	French<SPACE> Bonjour<SPACE>	1	

Lexicon Term	Generated Regex	Text to Match	Normalized Version of Text to Match	Match Count	Notes
Bonjour	(^|\s)(Bonjour)\s	French,Bonjour	French<SPACE>Bonjour<SPACE>	1	
Bonjour	(^|\s)(Bonjour)\s	French ,Bonjour	French <SPACE>Bonjour<SPACE>	1	
Bonjour	(^|\s)(Bonjour)\s	French, Bonjour.	French<SPACE> Bonjour<SPACE>	1	
devil's	(^|\s)(devil\s*'\s*s)\s	devil's advocate	devil's advocate<SPACE>	1	
devil'	(^|\s)(devil\s*')\s	devil's advocate	devil's advocate<SPACE>	0	The match doesn't conform to the token boundary.
devil'	(^|\s)(devil\s*')\s	devil'	devil<SPACE>	0	Any `'` characters are removed from the text by normalization.
\$5	(^|\s)(\$5)\s	$5	$5<SPACE>	1	

Regex Token-Based v3

Default Compile Mode	No
Use	Used for Lexicons that accept regex terms and are being applied to languages that use whitespace to separate tokens.
Limitations	This Compile Mode doesn't match on most punctuation characters; instead, they act as token separators.
Preprocessing	Follows the general preprocessing rules outlined in Preprocessing.
Grammar	Regex

The following table includes matches using this Compile Mode:

Lexicon Term	Generated Regex	Text to Match	Normalized Version of Text to Match	Match Count	Notes
Bonjour	(^|\s)(Bonjour)\s	French, Bonjour	French<SPACE> Bonjour<SPACE>	1	
Bonjour	(^|\s)(Bonjour)\s	French,Bonjour	French<SPACE>Bonjour<SPACE>	1	
Bonjour	(^|\s)(Bonjour)\s	French ,Bonjour	French <SPACE>Bonjour<SPACE>	1	
Bonjour	(^|\s)(Bonjour)\s	French, Bonjour.	French<SPACE> Bonjour<SPACE>	1	
devil's	(^|\s)(devil\s*'\s*s)\s	devil's advocate	devil's advocate<SPACE>	1	
devil'	(^|\s)(devil\s*')\s	devil's advocate	devil's advocate<SPACE>	0	The match doesn't conform to the token boundary.
devil'	(^|\s)(devil\s*')\s	devil'	devil<SPACE>	0	Any `'` characters are removed from the text by normalization.
\$5	(^|\s)(\$5)\s	$5	$5<SPACE>	1	

Natural Language Character-Based v2

Default Compile Mode	No
Use	Used for Lexicons that support Natural Language terms and where matches over character boundaries are preferred. For example, generating character-level matches on languages without whitespace.

Limitations	This Compile Mode doesn't apply any preprocessing. It uses character counts to calculate the window size `n` in `NEAR{n}` and `FOLLOWEDBY{n}` expressions. Using `*` with this Compile Mode could result in an indefinite number of matches on long text.
Preprocessing	**Important** No preprocessing is applied when using this Compile Mode.
Grammar	Natural Language

The following table includes matches using this Compile Mode:

Lexicon Terms	Text to Match	Match Count	Notes
AB FOLLOWEDBY{8} CD	ABUVWXCDYZ	1	
AB FOLLOWEDBY{4} CD	ABUVWXCDYZ	1	
AB FOLLOWEDBY{3} CD	AB1CD	1	
AB FOLLOWEDBY{3} CD	AB12CD	1	
AB FOLLOWEDBY{3} CD	AB123CD	1	
AB FOLLOWEDBY{3} CD	AB1234CD	0	
A B FOLLOWEDBY{8} C D	A B U V W X C D Y Z	0	FOLLOWEDBY distance is measured in terms of characters.
A B FOLLOWEDBY{8} C D	ABUVWXCDYZ	0	No spaces in the text to match.
ABC	Good morning ABC	1	
ABC	Good morning A B C	0	No direct matches with ABC because of spaces in the text.
Ng Ho Hai Dou Kong	Ng Ho Hai Dou Kong	1	
alpha*	alpha	1	

Lexicon Terms	Text to Match	Match Count	Notes
al?ha	alpha	1	
al*a	alpha	1	

Regex Character-Based v2

Default Compile Mode	No
Use	Used for Lexicons that accept regex terms and where matches over character boundaries are preferred. For example, generating character-level matches on languages without whitespace.
Limitations	This Compile Mode doesn't apply any preprocessing.
Preprocessing	**Important** No preprocessing is applied when using this Compile Mode.
Grammar	Regex

Some example matches for this Compile Mode are as follows:

Lexicon Terms	Text to Match	Match Count	Notes
AB	ABCDEFGHIJKLDEMNOP	1	
CD	ABCDEFGHIJKLDEMNOP	1	
GHI	ABCDEFGHIJKLDEMNOP	1	
DE	ABCDEFGHIJKLDEMNOP	2	
AB	XXXABABXXXAB	3	
alpha	alphas	1	
alphas	alpha	0	
ABC	Good morning ABC	1	