Text Data Processing Extraction Customization Guide

■ SAP BusinessObjects Data Services 4.1 (14.1.0)

2012-12-12

Contents

Introduction

1.1 Overview of This Guide

Welcome to the *Extraction Customization Guide*.

SAP BusinessObjects Data Services text data processing software enables you to perform extraction processing and various types of natural language processing on unstructured text.

The two major features of the software are linguistic analysis and extraction. Linguistic analysis includes natural-language processing (NLP) capabilities, such as segmentation, stemming, and tagging, among other things. Extraction processing analyzes unstructured text, in multiple languages and from any text data source, and automatically identifies and extracts key entity types, including people, dates, places, organizations, or other information, from the text. It enables the detection and extraction of activities, events and relationships between entities and gives users a competitive edge with relevant information for their business needs.

1.1.1 Who Should Read This Guide

This guide is written for dictionary and extraction rule writers. Users of this guide should understand extraction concepts and have familiarity with linguistic concepts and with regular expressions.

This documentation assumes the following:

- You understand your organization's text analysis extraction needs.

1.1.2 About This Guide

This guide contains the following information:

- Overview and conceptual information about dictionaries and extraction rules.
- How to create, compile, and use dictionaries and extraction rules.
- Examples of sample dictionaries and extraction rules.

- Best practices for writing extraction rules.

Using Dictionaries

A dictionary in the context of the extraction process is a user-defined repository of entities. It can store customized information about the entities your application must find. You can use a dictionary to store name variations in a structured way that is accessible through the extraction process. A dictionary structure can also help standardize references to an entity.

Dictionaries are language-independent. This means that you can use the same dictionary to store all your entities and that the same patterns are matched in documents of different languages.

You can use a dictionary for:

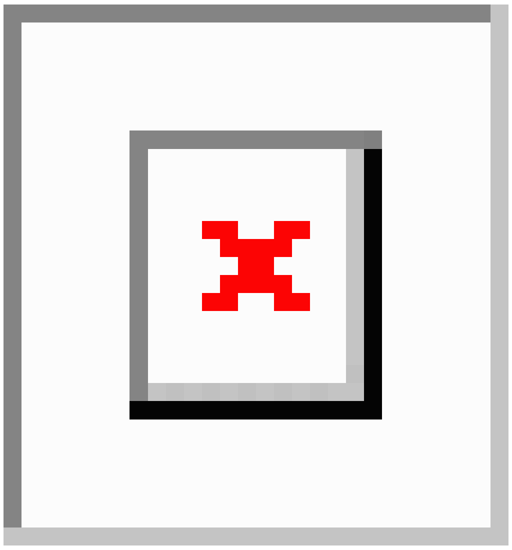
- name variation management
- disambiguation of unknown entities
- control over entity recognition

2.1 Entity Structure in Dictionaries

This section examines the entity structure in dictionaries. A dictionary contains a number of user-defined entity types, each of which contains any number of entities. For each entity, the dictionary distinguishes between a standard form name and variant names:

- Standard form name–The most complete or precise form for a given entity. For example, **United States of America** might be the standard form name for that country. A standard form name can have one or more variant names (also known as source form) embedded under it.
- Variant name–Less standard or complete than a standard form name, and it can include abbreviations, different spellings, nicknames, and so on. For example, **United States**, **USA** and **US** could be variant names for the same country. In addition, a dictionary lets you assign variant names to a type. For example, you might define a variant type `ABBREV` for abbreviations.

  The following figure shows a graphical representation of the dictionary hierarchy and structure of a dictionary entry for **United Parcel Service of America, Inc:**



The real-world entity, indicated by the circle in the diagram, is associated with a standard form name and an entity type `ORGANIZATION` and subtype `COMMERCIAL`. Under the standard form name are name variations, one of which has its own type specified. The dictionary lookup lets you get the standard form and the variant names given any of the related forms.

2.1.1 Generating Predictable Variants

The variants United Parcel Service and United Parcel Service of America, Inc. are predictable, and more predictable variants can be generated by the dictionary compiler for later use in the extraction process. The dictionary compiler, using its variant generate feature, can programmatically generate certain predictable variants while compiling a dictionary.

Variant generation works off of a list of designators for entities in the entity type `ORGANIZATION` in English. For instance, Corp. designates an organization. Variant generation in languages other than English covers the standard company designators, such as AG in German and SA in French. The variant generation facility provides the following functionality:

- Creates or expands abbreviations for specified designators. For example, the abbreviation Inc. is expanded to Incorporated, and Incorporated is abbreviated to Inc., and so on.
- Handles optional commas and periods.
- Makes optional such company designators as Inc, Corp. and Ltd, as long as the organization name has more than one word in it.

For example, variants for Microsoft Corporation can include:

- Microsoft Corporation
- Microsoft Corp.
- Microsoft Corp

Single word variant names like Microsoft are not automatically generated as variant organization names, since they are easily misidentified. One-word variants need to be entered into the dictionary individually. Variants are not enumerated without the appropriate organization designators.

Note:
Variant generation is supported in English, French, German, and Spanish.

Related Topics

• Adding Standard Variant Types

2.1.2 Custom Variant Types

You can also define custom variant types in a dictionary. Custom variant types can contain a list of variant name pre-modifiers and post-modifiers for a standard form name type. For any variant names of a standard form name to be generated, it must match at least one of the patterns defined for that custom variant type.

A variant generation definition can have one or more patterns. For each pattern that matches, the defined generators are invoked. Patterns can contain the wildcards `*` and `?`, that match zero-or-more and a single token respectively. Patterns can also contain one or more capture groups. These are sub-patterns that are enclosed in brackets. The contents of these capture groups after matching are copied into the generator output when referenced by its corresponding placeholder (if any). Capture groups are numbered left to right, starting at 1. A capture group placeholder consists of a backslash, followed by the capture group number.

The pattern always matches the entire string of the standard form name and never only part of that string. For example,

```
<define-variant_generation type="ENUM_TROOPS" >
    <pattern string="(?) forces" >
        <generate string="\1 troops" />
        <generate string="\1 soldiers" />
        <generate string="\1 Army" />
        <generate string="\1 military" />
        <generate string="\1 forces" />
    </pattern>
</define-variant_generation>
```

In the above example this means that:

- The pattern matches forces preceded by one token only. Thus, it matches Afghan forces, but not U.S. forces, as the latter contains more than one token. To capture variant names with more than one token, use the pattern `(*)` forces.
- The single capture group is referenced in all generators by its index: `\1`. The generated variant names are Afghan troops, Afghan soldiers, Afghan Army, Afghan military, and Afghan forces. In principle you do not need the last generator, as the standard form name already matches those tokens.

The following example shows how to specify the variant generation within the dictionary source:

```
<entity_name standard_form="Afghan forces">
    <variant_generation type="ENUM_TROOPS" /> \
    <variant name="Afghanistan's Army" />
</entity_name>
```

Note:
Standard variants include the base text in the generated variant names, while custom variants do not.

Related Topics

- Adding Custom Variant Types

2.1.3 Entity Subtypes

Dictionaries support the use of entity subtypes to enable the distinction between different varieties of the same entity type. For example, to distinguish leafy vegetables from starchy vegetables.

To define an entity subtype in a dictionary entry, add an `@` delimited extension to the category identifier, as in `VEG@STARCHY`. Subtyping is only one-level deep, so `TYPE@SUBTYPE@SUBTYPE` is not valid.

Related Topics

- Adding an Entity Subtype

2.1.4 Variant Types

Variant names can optionally be associated with a type, meaning that you specify the type of variant name. For example, one specific type of variant name is an abbreviation, `ABBREV`. Other examples of variant types that you could create are `ACRONYM`, `NICKNAME`, or `PRODUCT-ID`.

2.1.5 Wildcards in Entity Names

Dictionary entries support entity names specified with wildcard pattern-matching elements. These are the Kleene star (`"*"`) and question mark (`"?"`) characters, used to match against a portion of the input string. For example, either "`* University`" or "`? University`" might be used as the name of an entity belonging to a custom type `UNIVERSITY`.

These wildcard elements must be restricted to match against only part of the input buffer. Consider a pattern "`Company *`" which matches at the beginning of a 500 KB document. If unlimited matching were allowed, the `*` wildcard would match against the document's remaining 499+ KB.

Note:
Using wildcards in a dictionary may affect the speed of entity extraction. Performance decreases proportionally with the number of wildcards in a dictionary. Use this functionality keeping potential performance degradations in mind.

2.1.5.1 Wildcard Definitions

The `*` and `?` wildcards are described as follows, given a sentence:

- `*` matches any number of tokens greater than or equal to zero within a sentence.
- `?` matches only one token within a sentence.

A token is an independent piece of a linguistic expression, such as a word or a punctuation. The wildcards match whole tokens only and not sub-parts of tokens. For both wildcards, any tokens are eligible to be matching elements, provided the literal (fixed) portion of the pattern is satisfied.

2.1.5.2 Wildcard Usage

Wildcard characters are used to specify a pattern, normally containing both literal and variable elements, as the name of an entity. For instance, consider this input:

I once attended Stanford University, though I considered Carnegie Mellon University.

Consider an entity belonging to the category `UNIVERSITY` with the variant name "`* University`". The pattern will match any sentence ending with "`University`".

If the pattern were "`? University`", it would only match a single token preceding "`University`" occurring as or as a part of a sentence. Then the entire string "Stanford University" would match as intended. However, for "Carnegie Mellon University", it is the substring "Mellon University" which would match: "Carnegie" would be disregarded, since the question mark matches one token at most–and this is probably not the intended result.

If several patterns compete, the extraction process returns the match with the widest scope. Thus if a competing pattern "`* University`" were available in the previous example, "Carnegie Mellon University" would be returned, and "Mellon University" would be ignored.

Since `*` and `?` are special characters, "`escape`" characters are required to treat the wildcards as literal elements of fixed patterns. The back slash "`\`" is the `escape` character. Thus "`\*`" represents the literal asterisk as opposed to the Kleene star. A back slash can itself be made a literal by writing "`\\`".

Note:
Use wildcards when defining variant names of an entity instead of using them for defining a standard form name of an entity.

Related Topics

- Adding Wildcard Variants

2.2 Creating a Dictionary

To create a dictionary, follow these steps:

1. Create an **XML** file containing your content, formatted according to the dictionary syntax.
2. Run the dictionary compiler on that file.

   Note:
   For large dictionary source files, make sure the memory available to the compiler is at least five times the size of the input file, in bytes.

Related Topics

- Dictionary XSD
- Compiling a Dictionary

2.3 Dictionary Syntax

2.3.1 Dictionary XSD

The syntax of a dictionary conforms to the following XML Schema Definition (`XSD`). When creating your custom dictionary, format your content using the following syntax, making sure to specify the encoding if the file is not `UTF-8`.

```
<?xml version="1.0" encoding="UTF-8"?>
!--
Copyright 2010 SAP AG. All rights reserved.

SAP, R/3, SAP NetWeaver, Duet, PartnerEdge, ByDesign, SAP Business ByDesign,
and other SAP products and services mentioned herein as well as their
respective logos are trademarks or registered trademarks of SAP AG in
Germany and other countries.
```

```
Business Objects and the Business Objects logo, BusinessObjects, Crystal
Reports, Crystal Decisions, Web Intelligence, Xcelsius, and other Business
Objects products and services mentioned herein as well as their respective
logos are trademarks or registered trademarks of Business Objects S.A. in the
United States and in other countries.

Business Objects is an SAP company.

All other product and service names mentioned are the trademarks of their
respective companies. Data contained in this document serves informational
purposes only. National product specifications
may vary.

These materials are subject to change without notice. These materials are
provided by SAP AG and its affiliated companies ("SAP Group") for informational
purposes only, without representation or warranty of any kind, and SAP Group
shall not be liable for errors or omissions with respect to the materials.
The only warranties for SAP Group products and services are those that are set
forth in the express warranty statements accompanying such products and
services, if any. Nothing herein should be construed as constituting an
additional warranty.
--
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:dd="http://www.sap.com/ta/4.0"
  targetNamespace="http://www.sap.com/ta/4.0"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

 <xsd:element name="dictionary">
  <xsd:complexType>
   <xsd:sequence maxOccurs="unbounded">
    <xsd:element ref="dd:define-variant_generation" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="dd:entity_category" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="entity_category">
  <xsd:complexType>
   <xsd:sequence maxOccurs="unbounded">
    <xsd:element ref="dd:entity_name"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="entity_name">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="dd:variant" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="dd:query_only" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="dd:variant_generation" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="standard_form" type="xsd:string" use="required"/>
   <xsd:attribute name="uid" type="xsd:string" use="optional"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="variant">
  <xsd:complexType>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="type" type="xsd:string" use="optional"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="query_only">
  <xsd:complexType>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="type" type="xsd:string" use="optional"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="variant_generation">
  <xsd:complexType>
   <xsd:attribute name="type" type="xsd:string" use="required"/>
   <xsd:attribute name="language" type="xsd:string" use="optional" default="english"/>
   <xsd:attribute name="base_text" type="xsd:string" use="optional"/>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="define-variant_generation">
  <xsd:complexType>
   <xsd:sequence maxOccurs="unbounded">
    <xsd:element ref="dd:pattern"/>
   </xsd:sequence>
   <xsd:attribute name="type" type="xsd:string" use="required"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="pattern">
  <xsd:complexType>
   <xsd:sequence maxOccurs="unbounded">
    <xsd:element ref="dd:generate"/>
   </xsd:sequence>
   <xsd:attribute name="string" type="xsd:string" use="required"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="generate">
  <xsd:complexType>
   <xsd:attribute name="string" type="xsd:string" use="required"/>
  </xsd:complexType>
 </xsd:element>

</xsd:schema>
```

The following table describes each element and attribute of the dictionary XSD.

Element	Attributes and Description	
`dictionary`	This is the root tag, of which a dictionary may contain only one. Contains one or more embedded `entity_category` elements.	
`entity_category`	The category (type) to which all embedded entities belong. Contains one or more embedded `entity_name` elements. Must be explicitly closed.	
	`name`	The name of the category, such as PEOPLE, COMPANY, PHONE NUMBER, and so on. Note that the entity category name is case sensitive.

Element	Attributes and Description	
`entity_name`	A named entity in the dictionary. Contains zero or more of the elements `variant`, `query_only` and `variant_generation`. Must be explicitly closed.	
	`standard_form`	The standard form of the `entity_name`. The standard form is generally the longest or most common form of a named entity. The `standard_form` name must be unique within the `entity_category` but not within the `dictionary`.
	`uid`	A user-defined ID for the standard form name. This is an optional attribute.
`variant`	A variant name for the entity.The variant name must be unique within the `entity_name`. Need not be explicitly closed.	
	`name`	[Required] The name of the variant.
	`type`	[Optional] The type of variant, generally a subtype of the larger `entity_category`.
`query_only`		
	`name`	
	`type`	

Element	Attributes and Description	
`variant_generation`	Specifies whether the dictionary should automatically generate predictable variants. By default, the standard form name is used as the starting point for variant generation. Need not be explicitly closed.	
	`language`	[Optional] Specifies the language to use for standard variant generation, in lower case, for example, "english". If this option is not specified in the dictionary, the language specified with the compiler command is used, or it defaults to English when there is no language specified in either the dictionary or the compiler command.
	`type`	[Required] Types supported are standard or the name of a custom variant generation defined earlier in the dictionary.
	`base_text`	[Optional] Specifies text other than the standard form name to use as the starting point for the computation of variants.
`define-variant_generation`	Specifies custom variant generation.	
`pattern`	Specifies the pattern that must be matched to generate custom variants.	
`generate`	Specifies the exact pattern for custom variant generation within each generate tag.	

Related Topics

- Adding Custom Variant Types
- Formatting Your Source

2.3.2 Guidelines for Naming Entities

This section describes several guidelines for the format of standard form and variant names in a dictionary:

- You can use any part-of-speech (word class).
- Use only characters that are valid for the specified encoding.
- The symbols used for wildcard pattern matching, "`?`" and "`*`", must be escaped using a back slash character ("`\`") .
- Any other special characters, such as quotation marks, ampersands, and apostrophes, can be escaped according to the `XML` specification.

The following table shows some such character entities (also used in `HTML`), along with the correct syntax:

Character	Description	Dictionary Entry
`<`	Less than (`<`) sign	`&lt;`
`>`	Greater than (`>`) sign	`&gt;`
`&`	Ampersand (`&`) sign	`&amp;`
`"`	Quotation marks (`"`)	`&quot;`
`'`	Apostrophe (`'`)	`&apos;`

2.3.3 Character Encoding in a Dictionary

A dictionary supports all the character encodings supported by the **Xerces-C XML** parser. If you are creating a dictionary to be used for more than one language, use an encoding that supports all required languages, such as **UTF-8**. For information on encodings supported by the**Xerces-C XML** parser, see http://xerces.apache.org/xerces-c/faq-parse-3.html#faq-16.

The default input encoding assumed by a dictionary is UTF-8. Dictionary input files that are not in **UTF-8** must specify their character encoding in an `XML` directive to enable proper operation of the configuration file parser, for example:

```
<?xml version="1.0" encoding="UTF-16" ?>.
```

If no encoding specification exists, **UTF-8** is assumed. For best results, always specify the encoding.

Note:
CP-1252 must be specified as `windows-1252` in the **XML** header element. The encoding names should follow the **IANA-CHARSETS** recommendation.

2.3.4 Dictionary Sample File

Here is a sample dictionary file.

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="ORGANIZATION@COMMERCIAL">
        <entity_name standard_form="United Parcel Service of America, Incorporated">
            <variant name="United Parcel Service" />
            <variant name="U.P.S." type="ABBREV" />
            <variant name="UPS" />
            <variant_generation type="standard" language="english" />
        </entity_name>
    </entity_category>
</dictionary>
```

Related Topics

• Entity Structure in Dictionaries

2.3.5 Formatting Your Source

Format your source file according to the dictionary XSD. The source file must contain sufficient context to make the entry unambiguous. The required tags for a dictionary entry are:

- `entity_category`
- `entity_name`

Others can be mentioned according to the desired operation. If tags are already in the target dictionary, they are augmented; if not, they are added. The add operation never removes tags, and the remove operation never adds them.

Related Topics

• Dictionary XSD

2.3.6 Working with a Dictionary

This section provides details on how to update your dictionary files to add or remove entries as well as update existing entries.

2.3.6.1 Adding an Entity

To add an entity to a dictionary:

- Specify the entity's standard form under the relevant entity category, and optionally, its variants.

  The example below adds two new entities to the ORGANIZATION@COMMERCIAL category:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="ORGANIZATION@COMMERCIAL">
        <entity_name standard_form="Seventh Generation Incorporated">
            <variant name="Seventh Generation"/>
            <variant name="SVNG"/>
        </entity_name>
        <entity_name standard_form="United Airlines, Incorporated">
            <variant name="United Airlines, Inc."/>
            <variant name="United Airlines"/>
            <variant name="United"/>
        </entity_name>
    </entity_category>
</dictionary>
```

2.3.6.2 Adding an Entity Type

To add an entity type to a dictionary:

- Include a new `entity_category` tag

  For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="YOUR_ENTITY_TYPE">
        ...
    </entity_category>
</dictionary>
```

2.3.6.3 Adding an Entity Subtype

To add an entity subtype to a dictionary:

- include a new `entity_category` tag, using an @ delimited extension to specify the subtype.

  For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="VEG@STARCHY">
        ...
    </entity_category>
</dictionary>
```

2.3.6.4 Adding Variants and Variant Types

To add variants for existing entities:

- Include a `variant` tag under the entity's standard form name. Optionally, indicate the type of variant.

  For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="ORGANIZATION@COMMERCIAL">
        <entity_name standard_form="Apple Computer, Inc.">
            <variant name="Job's Shop" type="NICKNAME" />
        </entity_name>
    </entity_category>
</dictionary>
```

2.3.6.5 Adding Standard Variant Types

To add standard variant types,

- Include a `variant_generation` tag in an entity's definition.

  For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="ORGANIZATION@COMMERCIAL">
        <entity_name standard_form="Seventh Generation Inc">
            <variant_generation type="standard" language="english" />
        </entity_name>
    </entity_category>
</dictionary>
```

If you want variants generated for both standard form and variant names, use more than one `variant_generation` tag.

In the `language` attribute, specify the language for which variant generation applies; standard variant generations are language dependent. Variant generation is supported in English, French, German and Spanish.

2.3.6.6 Adding Custom Variant Types

To add custom variant types,

- Define a name with the list of variant generations.

  For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <define-variant_generation type="ENUM_INC" >
    <pattern string="(*) Inc" >
    <generate string="\1 Inc" />
    <generate string="\1, Inc" />
    <generate string="\1 Incorporated" />
    <generate string="\1, Incorporated" />
    </pattern>
    </define-variant_generation>

    <entity_category name="ORGANIZATION@COMMERCIAL">
    <entity_name standard_form="Seventh Generation Inc"
        <variant_generation type="ENUM_INC" />
        <variant_generation type="ENUM_INC" base_text="7th Generation Inc" />
        <variant_generation type="ENUM_INC" base_text="Seven Generation Inc" />
            <variant name="7th Generation Inc" />
            <variant name="Seven Generation Inc" />
    </entity_name>
    </entity_category>
</dictionary>
```

  The example should match the following expressions, with "Seventh Generation Inc" as the standard form name:

  - Seventh Generation Inc
  - Seventh Generation, Inc
  - Seventh Generation Incorporated
  - Seventh Generation, Incorporated
  - 7th Generation Inc
  - 7th Generation, Inc
  - 7th Generation Incorporated
  - 7th Generation, Incorporated
  - Seven Generation Inc
  - Seven Generation, Inc

- Seven Generation Incorporated
- Seven Generation, Incorporated

The pattern string for the variant generation includes the following elements used specifically for custom variant generation types:

- Pattern-This is the content specified in the `<pattern>` tag, within parenthesis, typically a token wildcard, as in the example above. The content is applied on the standard form name of the entity, gets repeated in the variants, and can appear before and after the user-defined content, numbered left to right within the `generate` tag, as in the example below.

  Note:
  Custom variants generate patterns exactly as specified within each generate tag, therefore the static content itself is not generated unless you include a generate tag for that specific pattern, as indicated by the second pattern tag in the example below.

- User-defined generate strings—This is the content that changes as specified in each `generate` tag, as shown in the examples. This is literal content that cannot contain wildcards.

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="ORGANIZATION@COMMERCIAL">
    <entity_name standard_form="ABC Corporation of America">
    <define-variant_generation type="ENUM_CORP" >
        <pattern string="(*) Corpo (*)" >
            <generate string="\1 Corp \2" />
            <generate string="\1 Corp. \2" />
            <generate string="\1 Corpo \2" />
            <generate string="\1 Corporation \2" />
        </pattern>
        <pattern string="*">
            <generate string="\1" />
        </pattern>
    </define-variant_generation>
    </entity_name>
    </entity_category>
</dictionary>
```

Note:
The variants pattern must cover the entire standard form name, not a substring of it.

Related Topics
- Dictionary XSD

2.3.6.7 Adding Wildcard Variants

To add wildcard variants:

- Define a name with a wildcard in it.

For example:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary>
    <entity_category name="BANKS">
        <entity_name standard_form="American banks">
            <variant name="Bank of * America" />
        </entity_name>
    </entity_category>
    ...
</dictionary>
```

This wildcard entry matches entities like Bank of America, Bank of Central America, Bank of South America, and so on.

2.4 Compiling a Dictionary

You create a new dictionary or modify an existing one by composing an XML file containing expressions as per the dictionary syntax. To replace dictionary material, first delete the elements to be changed, then add replacements. When your source file is complete, you pass it as input to the dictionary compiler (`tf-ncc`). The dictionary compiler compiles a dictionary binary file from your **XML**-compliant source text.

Note:
For large dictionary source files, make sure the memory available to the compiler is at least five times the size of the input file, in bytes.

Related Topics

- Dictionary XSD
- Working with a Dictionary

2.4.1 Command-line Syntax for Compiling a Dictionary

The command line syntax to invoke the dictionary compiler is:

```
tf-ncc [options] <input filename>
```

where,

`[options]` are the following optional parameters.

`<input_file>` specifies the dictionary source file to be compiled. This argument is mandatory.

Syntax	Description
`-d <language_module_directory>`	Specifies the directory where the language modules are stored. This is a mandatory option. You must specify this option along with the language directory location. The default location for the language directory is, `../TextAnalysis/languages` relative to the `LINK_DIR/bin` directory.
`-o <output filename>`	The path and filename of the resulting compiled dictionary. If none is supplied the file `lxtf2.nc` is created in the current directory.
`-v`	Indicates verbose. Shows progress messages.
`-l <language>`	Specifies the default language for standard variant generation. If no language is specified in the tag or on the command line, *english* will be used. **Note:** Encoding must be specified by a `<?xml encoding=X>` directive at the top of the source file or it is assumed to be `utf-8`.
`-config_file <filename>`	Specifies the dictionary configuration file. The default configuration file `tf.nc-config` is located at `../TextAnalysis/languages` relative to the `LINK_DIR/bin` directory.
`-case_sensitive`	Generates case-sensitive variants. **Note:** If you include this command, you should include every variant of the word.
`-case_insensitive`	

Syntax	Description
	Generates case-insensitive variants. **Note:** Use caution when compiling a dictionary in case-insensitive mode as spurious entries may result. For instance, if either of the proper nouns May or Apple were listed in a case-insensitive dictionary, then the verb may and the fruit apple would be matched.
`-version`	Displays the compiler version.
`-h, -help, --help`	Prints a help message.

Note:
If you want to add dictionary entries, remove dictionary entries, or remove standard form names from a dictionary, you must modify your source file accordingly and recompile.

Related Topics

• Dictionary XSD

Using Extraction Rules

Extraction rules (also referred to as CGUL rules) are written in a pattern-based language that enables you to perform pattern matching using character or token-based regular expressions combined with linguistic attributes to define custom entity types.

You can create extraction rules to:

- Extract complex facts based on relations between entities and predicates (verbs or adjectives).
- Extract entities from new styles and formats of written communication.
- Associate entities such as times, dates, and locations, with other entities (entity-to-entity relations).
- Identify entities in unusual or industry-specific language. For example, use of the word crash in computer software versus insurance statistics.
- Capture facts expressed in new, popular vernacular. For example, recognizing sick, epic, and fly as slang terms meaning good.

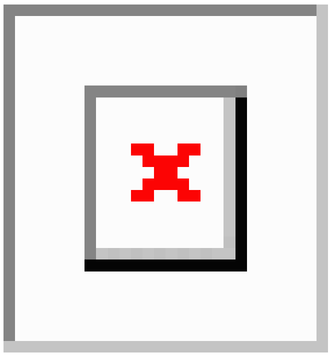
3.1 About Customizing Extraction

The software provides tools you can use to customize extraction by defining extraction rules that are specific to your needs.

To create extraction rules, you write patterns using regular expressions and linguistic attributes that define categories for the entities, relations, and events you need extracted. These patterns are written in CGUL (Custom Grouper User Language), a token-based pattern matching language. These patterns form rules that are compiled by the rule compiler (`tf-cgc`). The rule compiler checks CGUL syntax and logs any syntax errors.

Extraction rules are processed in the same way as pre-defined entities. It is possible to define entity types that overlap with pre-defined entities.

Once your rules are created, saved into a text file, and compiled into a binary `(.fsm)` file, you can test them using the Entity Extraction transform in the Designer.

Following diagram describes a basic workflow for testing extraction rules: 

Related Topics

- Compiling Extraction Rules
- Designer Guide: Transforms, Text Data Processing transforms, To add a text data processing transform to a data flow

3.2 Understanding Extraction Rule Patterns

With CGUL, you define extraction rules using character or token-based regular expressions combined with linguistic attributes. The extraction process does not extract patterns that span across paragraphs. Therefore, patterns expressed in CGUL represent patterns contained in one paragraph; not patterns that start in one paragraph and end in the next.

Tokens are at the core of the CGUL language. The tokens used in the rules correspond with the tokens generated by the linguistic analysis. Tokens express a linguistic expression, such as a word or punctuation, combined with its linguistic attributes. In CGUL, this is represented by the use of literal strings or regular expressions, or both, along with the linguistic attributes: part-of-speech (`POS`) and `STEM`

`STEM` is a base form– a word or standard form that represents a set of morphologically related words. This set may be based on inflections or derivational morphology.

The linguistic attributes supported vary depending on the language you use.

For information about the supported languages and about the linguistic attributes each language supports, refer to the *Text Data Processing Language Reference Guide*.

3.2.1 CGUL Elements

CGUL rules are composed of the elements described in the following table. Each element is described in more detail within its own section.

Element	Description
`CGUL` Directives	These directives define character classes (`#define`), subgroups (`#subgroup`), and facts (`#group`). For more information, see CGUL Directives .

Tokens	Tokens can include words in their literal form (cars) or regular expressions (car.*), their stem (car), their part-of-speech (`Nn`), or any of these elements combined:
	
	Tokens are delimited by angle brackets (`< >`) .
	
	`<car.*, POS:Nn>`
	
	`<STEM:fly, POS:V>`
	
	For more information, see Tokens.

Element	Description	
Operators	The following operators are used in building character patterns, tokens, and entities.	
	Iteration Operators	These include the following quantifier operators: `+`, `*`, `?`, `{ m}`, `{n,m}`.For more information, see Iteration Operators Supported in CGUL.
	Standard Operators	These include the following: • Character wildcard (`.`) • Alternation (`|`) • Escape character (`\`) • Character and string negation (`^` and `~`) • Subtraction (`-`) • Character Classifier (`\p{value}`) For more information, see, Standard Operators Valid in CGUL.
	Grouping and Containment Operators	

Element	Description	
		These include the following operators: `[range] , (item), {expression}`, where, • `[range]` defines a range of characters • `(item)` groups an expression together to form an item that is treated as a unit • `{expression}` groups an expression together to form a single rule, enabling the rule writer to wrap expressions into multiple lines For more information, see Grouping and Containment Operators Supported in CGUL.
	Expression Markers	These include the following markers: `[SN]`– Sentence `[NP]`–Noun phrase `[VP]`–Verb phrase `[CL]` and `[CC]`–Clause and clause container `[OD]`–Context `[TE]`–Entity `[UL]`and `[UC]`–Unordered list and contiguous unordered list `[P]`–Paragraph For more information, see Expression Markers Supported in CGUL.

Element	Description
	The following match filters can be used to specify whether a CGUL preceding token expression matches the longest or shortest pattern that applies. Match filters include: • `Longest match` • `Shortest match` • `List` (returns all matches) For more information, see Match Filters Supported in CGUL.
`Include` Directive	`#include` directives are used to include other CGUL source files and `.pdc` files. You must include CGUL source files and `.pdc` files before you use the extraction rules or predefined classes that are contained in the files you include. For more information, see Including Files in a Rule File.
`Lexicon` Directive	`#lexicon` directives are used to include the contents of a dictionary file that contains a list of single words, delimited by new lines. For more information, see Including a Dictionary in a Rule File.
Comments	Comments are marked by an initial exclamation point (`!`). When the compiler encounters a `!` it ignores the text that follows it on the same line.

3.2.2 CGUL Conventions

CGUL rules must follow the following conventions:

- Rules are case-sensitive by default. However, you can make rules or part of rules case insensitive by using character classifiers.
- A blank space is required after `#define`, `#subgroup`, `#group`, `#include`, `#lexicon`, and between multiple key-value pairs. Otherwise, blank spaces are rejected unless preceded by the escape character.
- Names of CGUL directives (defined by `#define`, `#subgroup`, or `#group`) must be in alphanumeric `ASCII` characters with underscores allowed.
- You must define item names before using them in other statements.

Related Topics

- Character Classifier (\p)

3.3 Including Files in a Rule File

You can include CGUL source files and `.pdc` files anywhere within your rules file. However, the entry point of an `#include` directive should always precede the first reference to any of its content.

The syntax for the included files is checked and separate error messages are issued if necessary.

Note:
The names defined in included files cannot be redefined.

Syntax

```
#include <filename>
#include "filename"
```

where *`filename`* is the name of the CGUL rules file or `.pdc` file you want to include. You can use an absolute or relative path for the file name, but it is recommended that you use an absolute path.

Note:
The absolute path is required if:

- The input file and the included file reside in different directories.
- The input file is not stored in the directory that holds the compiler executable.
- The file is not in the current directory of the input file.

3.3.1 Using Predefined Character Classes

The extraction process provides predefined character and token classes for each language supported by the system. Both character classes and token classes are stored in the `<language>.pdc` files.

To use these classes, use the`#include` statement to include the `.pdc` file in your rule file.

3.4 Including a Dictionary in a Rule File

You can include a dictionary within your rules file. However, the entry point of a `#lexicon` directive should always precede the first reference to its content.

The dictionary file consists of single words separated by new lines. The compiler interprets the contents of a dictionary as a `#define` directive with a rule that consists of a bracketed alternation of the items listed in the file.

Syntax

```
#lexicon name "filename"
```

where,

`name` is the CGUL name of the dictionary.

`filename` is the name of the file that contains the dictionary.

Example

```
#lexicon FRUITLIST "myfruits.txt"
...
#group FRUIT: <STEM:%(FRUITLIST)>
```

Description

In this example, the dictionary is compiled as a `#define` directive named `FRUITLIST` and is contained in a file called `myfruits.txt`. Later in the rule file the dictionary is used in the `FRUIT` group. If `myfruits.txt` contains the following list:

```
apple
orange
banana
peach
strawberry
```

The compiler interprets the group as follows:

```
#group FRUIT: <STEM:apple|orange|banana|peach|strawberry>
```

Note:
A dictionary cannot contain entries with multiple words. In the preceding example, if `wild cherry` was included in the list, it would not be matched correctly.

3.5 CGUL Directives

CGUL directives define character classes (`#define`), tokens or group of tokens (`#subgroup`), and entity, event, and relation types (`#group`). The custom entities, events, and relations defined by the `#group` directive appear in the extraction output. The default scope for each of these directives is the sentence.

The relationship between items defined by CGUL directives is as follows:

- Character classes defined using the `#define` directive can be used within `#group` or `#subgroup`
- Tokens defined using the `#subgroup` directive can be used within `#group` or `#subgroup`
- Custom entity, event, and relation types defined using the `#group` directive can be used within `#group` or `#subgroup`

Related Topics

- CGUL Conventions
- Writing Directives
- Using the #define Directive
- Using the #subgroup Directive
- Using the #group Directive
- Using Items in a Group or Subgroup

3.5.1 Writing Directives

You can write directives in one line, such as:

```
#subgroup menu: (<mashed><potatoes>)|(<Peking><Duck>)|<tiramisu>
```

or, you can span a directive over multiple lines, such as:

```
#subgroup menu:
{
(<mashed><potatoes>)|
(<Peking><Duck>)|
<tiramisu>
}
```

To write a directive over multiple lines, enclose the directive in curly braces `{}`.

3.5.2 Using the #define Directive

The `#define` directive is used to denote character expressions. At this level, tokens cannot be defined. These directives represent user-defined character classes. You can also use predefined character classes.

Syntax

```
#define name: expression
```

where,

`name`– is the name you assign to the character class.

colon (`:`)– must follow the name.

`expression`– is the literal character or regular expression that represents the character class.

Example

```
#define ALPHA: [A-Za-z]
#define URLBEGIN: (www\.|http\:)
```

```
#define VERBMARK: (ed|ing)
#define COLOR: (red|blue|white)
```

Description

`ALPHA` represents all uppercase and lowercase alphabetic characters.

`URLBEGIN` represents either `www.` or `http:` for the beginning of a **URL** address

`VERBMARK` represents either `ed` or `ing` in verb endings

`COLOR` represents red, blue, or white

Note:
A `#define` directive cannot contain entries with multiple words. In the preceding example, if `navy blue` was included in the list, it would not be matched correctly.

Related Topics

• Using Predefined Character Classes

3.5.3 Using the #subgroup Directive

The `#subgroup` directive is used to define a group of one or more tokens. Unlike with `#group` directives, patterns matching a subgroup do not appear in the extraction output. Their purpose is to cover sub-patterns used within groups or other subgroups.

Subgroups make `#group` directives more readable. Using subgroups to define a group expression enables you to break down patterns into smaller, more precisely defined, manageable chunks, thus giving you more granular control of the patterns used for extraction.

In `#subgroup` and `#group` statements alike, all tokens are automatically expanded to their full format: `<literal, stem, POS>`

Note:
A rule file can contain one or more subgroups. Also, you can embed a subgroup within another subgroup, or use it within a group.

Syntax

```
#subgroup name:<expression>
```

where,

`name`– is the name you are assigning the token

colon (`:`)– must follow the name

`<expression>`– is the expression that constitutes the one or more tokens, surrounded by angle brackets `<>`, if the expression includes an item name, the item's syntax would be: `%(item)`

Example

```
#subgroup Beer: <Stella>|<Jupiler>|<Rochefort>
#subgroup BeerMod: %(Beer) (<Blonde>|<Trappist>)
#group BestBeer: %(BeerMod) (<Premium>|<Special>)
```

Description

The `Beer` subgroup represents specific brands of beer (Stella, Jupiler, Rochefort). The `BeerMod` subgroup embeds `Beer`, thus it represents any of the beer brands defined by `Beer`, followed by the type of beer (Blonde or Trappist). The `BestBeer` group represents the brand and type of beer defined by `BeerMod`, followed by the beer's grade (Premium or Special). To embed an item that is already defined by any other CGUL directive, you must use the following syntax: `%(`*`item`*`)`, otherwise the item name is not recognized.

Using the following input...

Beers in the market this Christmas include Rochefort Trappist Special and Stella Blonde Special.

...the sample rule would have these two matches:

- Rochefort Trappist Special
- Stella Blonde Special

3.5.4 Using the #group Directive

The `#group` directive is used to define custom facts and entity types. The expression that defines custom facts and entity types consists of one or more tokens. Items that are used to define a custom entity type must be defined as a token. Custom facts and entity types appear in the extraction output.

The `#group` directive supports the use of entity subtypes to enable the distinction between different varieties of the same entity type. For example, to distinguish leafy vegetables from starchy vegetables. To define a subtype in a `#group` directive add a `@` delimited extension to the group name, as in `#group VEG@STARCHY: <potatoes|carrots|turnips>`.

Note:
A rule file can contain one or more groups. Also, you can embed a group within another group.

Syntax

```
#group name: expression #group name@subtype: expression
#group name (scope="value"): expression
#group name (paragraph="value"): expression
#group name (key="value"): expression
#group name (key="value" key="value" key="value"): expression
```

The following table describes parameters available for the `#group` directive.

Note:
You can use multiple key-value pairs in a group, including for `paragraph` and `scope`.

Parameter	Description	Example
`name`	The name you assign to the extracted fact or entity	

`expression`	The expression that constitutes the entity type; the expression must be preceded by a colon (:)	`#group BallSports: <baseball> | <football> | <soccer>`
		
		`#group Sports: <cycling> | <boxing> | %(Ball Sports)`
		
		or
		
		`#group BodyParts: <head> | <foot> | <eye>`
		
		`#group Symptoms: %(Body Parts) (<ache>|<pain>)`
		
		In this example, the `Ball Sports` group represents sports played with a ball (baseball, football, soccer), while the `Sports` group represents cycling, boxing, or any of the sports defined by `BallSports`.

Parameter	Description	Example

`scope=`
 	
`"value"`
	An optional key-value pair that specifies the scope of the input to be interpreted by the pattern matcher. The value is either sentence or paragraph. When this key is not specified, the scope defaults to sentence.	`#group JJP (scope="paragraph"): <Jack> <>* <Jill>`
		
		Will match `Jack` followed by `Jill` anywhere within the same paragraph
		
		`#group JJS (scope="sentence"): <Jack> <>* <Jill>`
		
		and
		
		`#group JJ: <Jack> <>* <Jill>`
		
		Will match `Jack` followed by `Jill` anywhere within the same sentence.

`paragraph=`
 	
`"value"`
	An optional key-value pair that specifies which paragraphs to process. In this case, value represents a range of integers, plus the special symbol L to represent the last paragraph of the input.	`#group A (paragraph="[1]"): ...`
		
		`#group C (paragraph="[1-4]"): ...`
		
		`#group D (paragraph="[1-3, 6, 9]"): ...`
		
		`#group E (paragraph="[4-L]"): ...`
		
		In this example, each group processes the input as follows:
		• Group `A` processes the first paragraph only
		• Group `C` processes paragraphs 1 through 4
		• Group `D` processes paragraphs 1 through 3, paragraph 6, and paragraph 9
		• Group `E` processes the fourth through the last paragraph

Parameter	Description	Example
key= "*value*"	An optional key-value pair that represents any kind of user-defined key-value pair. In this case, *value* represents a user-defined value to be returned with each match on the group rule.	`#group Y (sendto="mjagger@acme.com"): ...` `#group Z (alert="Orange"): ...` In this example, each group returns the *value* in the key-value pair with each match on the group rule.

3.5.5 Using Items in a Group or Subgroup

You can use an item defined by any CGUL directive in a group or subgroup.

Syntax

You must precede the item name with the `%` operator and surround it with parenthesis. Also, the item must be defined as a token, or be part of a larger expression that is defined as a token.

```
%(item_name)
```

Example

```
#define LOWER: [a-z]
#define UPPER: [A-Z]
#subgroup INITCAP: <%(UPPER)%(LOWER)+>
#group MRPERSON: <Mr\.> %(INITCAP)
```

In this example, the items `UPPER` and `LOWER` are part of a larger expression that is defined as a token within the subgroup `INITCAP`. `INITCAP` is then used within a group, `MRPERSON`. `INITCAP` does not have to be declared a token within `MRPERSON` because it is already defined as a token in the `#subgroup` statement.

To use the item as a token you must surround it with angle brackets (<>) . However, once you define a token, you cannot surround the token name with angle brackets again. For example, the following `#group` statement is wrong, because `INITCAP` was already enclosed by `<>` in the `#subgroup` statement. In this case, an error message is issued:

```
#subgroup INITCAP: <%(UPPER)%(LOWER)+>
#group MRPERSON: <Mr\.> <%(INITCAP)>
```

3.6 Tokens

Tokens (also referred as syntactic units) are at the core of CGUL. They express an atomic linguistic expression, such as a word or punctuation, combined with its linguistic attributes: part-of-speech (`POS`) and `STEM`. CGUL uses tokens to represent the linguistic expressions to be matched.

3.6.1 Building Tokens

You can specify tokens that express a broad variety of patterns to help you define custom entity, event, and relation types and extract related entities. To build tokens, you use any of three optional fields: `string`, `STEM`, and `POS` (part-of-speech) tags. The `string` and `STEM` fields can use all valid CGUL operators to define patterns, while the `POS` field only accepts alternation and string negation.

Syntax

```
<string, STEM:stem, POS:"pos_tag">
<string, STEM:stem, POS:pos_tag>
```

where,

`string` can be a word, or a regular expression that represents a word pattern.

`stem` can be a word stem or a regular expression that represents a stem pattern.

`pos_tag` is a part-of-speech tag.

Note:
The part-of-speech tag can be expressed within or without quotation marks. The behavior is as follows:

- "*`pos_tag`*" (within quotation marks)– The `POS` value matches exactly. For example `<POS:"Adj">` matches only `Adj`, and `<POS: "Adj"|"Punct">` matches `Adj` or `Punct`.
- Each part-of-speech value requiring an exact match must be surrounded by quotes. Hence an expression such as `<POS:"Adj|Punct">` is syntactically invalid.
- *`pos_tag`* (no quotation marks)– The `POS` value includes the umbrella part-of-speech and all its expansions. for example `<POS:Adj>` matches `Adj`, and `Adj-Sg`, `Adj-Pl`, and so on.

Tokens conform to the following syntactic rules:

- Tokens must be delimited by angle brackets <>

  ```
  #subgroup BADDOG2: bulldog
  ```

  This is literally a character sequence and the string is not expanded with a `STEM` and a `POS`, therefore, it does not match the token bulldog. The proper notation is:

  ```
  #subgroup DOG2: <bulldog>
  ```

- Tokens are composed of three optional fields: `Literal`, `STEM`, and `POS`

  For example,

  ```
  <activat.+, STEM:activat.+, POS:V>
  ```

- The fields within the token are optional and are delimited by commas.

  Note:
  Fields that are not defined are expanded to the following defaults: .+ for literal, `ANYSTEM` for `STEM`, and `ANYPOS` for `POS`. Hence, they are assumed to be any possible value for that specific field.

  For example,

  ```
  <STEM:be, POS:V>
  ```

  means any token that has a stem be and is a verb

- `STEM` and `POS` must be written in all uppercase, followed by a colon (`:`), and separated by a comma.
- `POS` can be any part-of-speech tag.
- Blank spaces are ignored either within or between tokens, thus `<POS:V>` and `<POS: V>` are the same, and `<apple><tree>` and `<apple> <tree>` are the same.
- Items that are already defined as tokens cannot be surrounded by angle brackets (`<>`) when used as part of another definition.

  For example, the following `#group` statement is incorrect, and generates an error because *`COLOR`* is already defined as a token.

  ```
  #subgroup COLOR: <(red|white|blue)>
  ```

  ```
  #group FLAG_COLORS: <%(COLOR)>
  ```

  The correct statement would be:

  ```
  #subgroup COLOR: <(red|white|blue)>
  ```

  ```
  #group FLAG_COLORS: %(COLOR)
  ```

- You can refer to any token or a sequence of tokens by using the empty or placeholder token with the appropriate regular expression operator:
  - `<>` for any token
  - `<>*` for a series of zero or more tokens
  - `<>?` for zero or one token
  - `<>+` for a series of one or more tokens

- A sequence of related tokens, like "`German Shepherd`" needs each of its elements enclosed by token delimiters.

  For example, the following subgroup returns German Shepherd

  ```
  #subgroup DOG: <German><Shepherd>
  ```

  Whereas the following subgroup returns an error

  ```
  #subgroup DOG: <German Shepherd>
  ```

- Character classifiers do not operate in tokens that contain `STEM` or `POS` expressions, unless the classifier is assigned to the `STEM` value. However, operations on `POS` values are invalid.

Examples

`<car>`

- means: "`car STEM:`*`anystem`* `POS:`*`anypos`*"
- matches: all instances of **car**

`<ground, STEM: grind>`

- means: "`ground STEM:grind POS:`*`anypos`*"
- matches: The **ground** was full of stones. We took some and **ground** them to pieces.
- Only the second instance of the word **ground** matches the stem definition.

`<STEM: ground|grind>`

- means: "`.+ STEM:ground|grind POS:` *`anypos`*"
- matches: The **ground** was full of stones. We took some and **ground** them to pieces.
- Both instances of the word **ground** matches the stem definition.

`<POS: V>`

- means: "`.+ STEM:`*`anystem`* `POS:V`"
- matches: all verbs found in the input

`<POS: Adj|Nn>`

- means: "`.+ STEM:`*`anystem`* `POS:Adj|Nn`"
- matches: all adjectives and nouns found in the input

`<activat.+>`

- means: "`activat.+ STEM:`*`anystem`* `POS:`*`anypos`*"
- matches: **activation**, **activate**, **activator**, **activating**, **activated**, and so on.

`<STEM: cri.*>`

- means: "`.+ STEM: cri.+ POS:`*`anypos`*"
- matches: **crime**, **crimes**, **criminal**
- Note that it does not match: **cries** and **cried** because their stem is **cry**.

`<cri.*>` matches **crime**, **crimes**, **criminal** as well as **cries** and **cried**.

`<STEM: run, POS:V>`

- means: "`.+ STEM: run POS:V`"
- matches: all inflectional forms of the verb run such as **run**, **runs**, **running**, **ran**, but not the noun **run** as in a **5-mile run**.

Related Topics

- Text Data Processing Language Reference Guide: Part-of-Speech Support

3.7 Expression Markers Supported in CGUL

CGUL supports the expression markers as described in the following table.

Note:
All markers are matched following the shortest match principle. Also, all markers must be paired, with the exception of `[P]`.

Some expression markers, such as `[NP]`, `[VP]`, `[OD]`, and `[TE]`, can use key-value pairs to specify attributes, such as syntactic functions. When using key-value pairs, the following rules apply:

- Key-value pairs must include the value within double quotes (`key="value"`, `attribute="value1|value2|value3"`)
- Multiple key-value pairs are delimited by blank spaces only (`key1="value" key2="value1|value2" key3="value3|value4"`)

Operator	Description
Paragraph marker `([P] [/P])`	Specifies the beginning and end of a paragraph.
Sentence marker `([SN] [/SN])`	Specifies the beginning and end of a sentence.
Noun Phrase marker `([NP funct="`*value*`"] expr [/NP])`	Specifies the exact range of an expression `expr` that is a noun phrase.
Verb Phrase marker `([VP funct="`*value*`"] expr [/VP])`	Specifies the exact range of an expression `expr` that is a verb phrase.
Clause marker `([CL] expr [/CL])`	Specifies the exact range of the expression `expr` that is a clause
Clause container `([CC] expr [/CC])`	Matches the entire clause provided that the expression `expr` is matched somewhere inside that clause.

Operator	Description
Context (output) marker `(exL [OD name="`*`value`*`"] exp [/OD] exR)`	Specifies the pattern to be output by the extraction process. (Output Delimiter) **Note:** If the expression between the output delimiters allows zero tokens to match and the output is an empty string, the empty output is not displayed.
Entity marker `([TE name="`*`value`*`"] expr [/TE])`	Specifies the exact range of the expression `expr` to be an entity type or list of entity types.
Unordered list marker `([UL] expr1, expr2 , ..., exprN [/UL])`	Matches a set of expressions (`expr1`, `expr2`, and so on.) regardless of the order in which they match.
Unordered contiguous list marker `([UC] expr1, expr2 , ..., exprN [/UC])`	This is similar to the unordered list markers `[UL]` except for the additional restriction that all listed elements should form a contiguous string.

Related Topics
- Paragraph Marker [P]
- Sentence Marker [SN]
- Noun Phrase Marker [NP]
- Verb Phrase Marker [VP]
- Clause Marker [CL]
- Clause Container [CC]
- Context Marker [OD]
- Entity Marker [TE]
- Unordered List Marker [UL]
- Unordered Contiguous List Marker [UC]

3.7.1 Paragraph Marker [P]

Use the paragraph marker `[P] [/P]` to mark the beginning and end of a paragraph. These markers do not have to be paired.

In the following example, the expression matches any paragraph that begins with *`In sum`*.

```
[P]<In><sum><>+[/P]
```

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is in `#define` directives
- It is found inside `SN`, `CL`, `CC`, `NP`, `VP` and `TF` markers

3.7.2 Sentence Marker [SN]

Use the sentence marker `[SN] [/SN]` to mark the beginning and end of a sentence.

In the following example, the expression matches any sentence that has a form of **conclusion** as its first or second token:

```
#group CONCLUDE: [SN] < >? <STEM:conclusion> < >* [/SN]
```

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is in `#define` directives
- It is inside `CL`, `CC`, `NP`, `TE` and `VP` markers
- It is not used in pairs

3.7.3 Noun Phrase Marker [NP]

Use the noun phrase marker `[NP funct="`*`value`*`"] expr [/NP]` to specify the exact range of an expression that is a noun phrase, following the shortest match principle.

In the following example, the expression matches noun phrases that contain any form of the word weapon.

```
#group WEAPONS: [NP] < >* <STEM:weapon> [/NP]
```

The optional key `funct` specifies the syntactic function of the NP, either as subject (`Sub`), direct object (`Obj`), or predicate (`Pre`).

In the following example, the expression matches `Bears` in the first of the three sample phrases.

```
[NP funct="Sub"] <>* <STEM:bear> [/NP]
```

- In this sample, **Bears** is the subject.

  Bears eat everything.
- In this case, **bears** is the object.

  People love **bears**.
- In this case, **bears** is the predicate.

  Grizzlies are **bears**.

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is in `#define` statements
- It is inside other `NP`, `TE`, and `VP` markers
- It is not used in pairs

3.7.4 Verb Phrase Marker [VP]

Use the verb phrase marker `[VP funct="`*`value`*`"] expr [/VP]` to specify the exact range of an expression that is a verb phrase, following the shortest match principle.

Note:
The Verb Phrase contains only one token, the main verb of the clause.

In the following example, the expression matches verb phrases that contain any form of the verb love.

```
[VP] <STEM:love> [/VP]
```

The optional key `funct` specifies the syntactic function of the `VP`, as an active main verb (`MV`), a passive main verb (`MVP`), or as a copula verb (`MVC`).

In the following example, the expression matches eaten in the second of the three sample phrases.

```
[VP funct="MVP"] <> [/VP]
```

- In this sample, eat is the active main verb.

  Bears eat everything.
- In this sample, eaten is the passive main verb.

  People were eaten by bears.
- In this sample, are is the copula verb.

Grizzlies are bears.

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- It is inside other `VP`, `TE`, and `NP` markers
- It is not used in pairs

3.7.5 Clause Marker [CL]

Use the clause marker `[CL] expr [/CL]` to specify the exact range of an expression that is a clause, following the shortest match principle. The clause marker currently supports the following languages: English, French, Spanish, German, and Simplified Chinese.

In the following example, any clause that starts with **in conclusion** is matched.

```
#group SUMMARY: [CL] <in> <conclusion> < >* [/CL]
```

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- It is inside other `CL`, `CC`, `NP`, `TE`, and `VP` markers
- It is not used in pairs

3.7.6 Clause Container [CC]

The clause container marker `[CC] expr [/CC]` matches the entire clause provided that the expression `expr` is matched somewhere within that clause. The clause container marker currently supports the following languages: English, French, Spanish, German, and Simplified Chinese.

In the following example, any clause that contains forms of **euro** and **dollar** is matched.

```
#group CURRENCY: [CC] <STEM:euro> < >* <STEM:dollar> [/CC]
```

Using `[CL]` to specify that an expression can appear anywhere in a clause achieves the same result as using the clause container `[CC]` operator, in other words:

```
[CC] expr [/CC]
```

achieves the same results as:

```
[CL] < >* expr < >* [/CL]
```

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- It is inside other `CL`, `CC`, `NP`, `TE`, and `VP` markers
- It is not used in pairs

3.7.7 Context Marker [OD]

Use the context marker `OD` (Output Delimiter) in `exL [OD funct="` *`value`* `"] exp [/OD] exR` to specify which part of the matched string is shown in the resulting output.

In this case, `exL` (that is, left), `exp,` and `exR` (that is, right) are all matched, but only `exp` (between the `[OD]` markers) is shown in the output.

In the following example, any proper nouns preceded by either `Sir` , `Mr` , or `Mister` are matched. The resulting pattern is just the proper noun.

```
#group Name: <Sir|Mr|Mister> [OD] <POS: Prop> [/OD]
```

Optionally, you can add a label to the output, as shown in the following examples. The extraction process picks this label up as the entity name for the string that is output.

```
#group Name: <Sir|Mr|Mister> [OD LastName] <POS: Prop> [/OD]
#group Name: <Sir|Mr|Mister> [OD name="LastName"] <POS: Prop> [/OD]
```

Note:
If the expression between the `OD` markers allows zero tokens to match and the output is an empty string, the empty output is not displayed.

You can use multiple context markers to match discontinuous patterns.

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- It is inside unordered lists `[UL]` and unordered contiguous lists `[UC]`
- It is inside other context markers `[OD]`
- It is not used in pairs

Related Topics

- Writing Extraction Rules Using Context Markers

3.7.8 Entity Marker [TE]

Use the entity marker `[TE name] exp [/TE]` to specify the exact range of an expression that is a pre-defined entity of type `name`.

In the following examples, the expressions match any entity of type `PERSON` with **Bush** as its last element.

```
#group President: [TE PERSON] < >* <Bush> [/TE]
#group President: [TE name="PERSON"] < >* <Bush> [/TE]
```

You can also specify lists of entities, as follows:

```
[TE  PERSON|ORGANIZATION@COMMERCIAL]
[TE  name="PERSON|ORGANIZATION@COMMERCIAL"]
```

Finally, you can specify subtyped category names, as follows

```
#group X: [TE VEHICLE@LAND] <>+ [/TE]
#group Y: [TE VEHICLE] <>+ [/TE]
```

Where Group X matches only the subtyped entities, and Group Y matches all **VEHICLE** entities, regardless of their subtyping.

This marker is invalid when:

- It is inside the range operator `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- A specified value is not an existing entity
- It is not used in pairs

3.7.9 Unordered List Marker [UL]

Use the unordered list marker `[UL]expr1, expr2, ..., exprN [/UL]` to match a set of comma delimited expressions regardless of the order in which they match.

Note:
This marker impacts processing time, use sparingly.

The following example matches `Shakespeare` , `Faulkner` and `Hemingway` if found in any order. Nesting the unordered list within a clause container `[CC]` operator limits the occurrence of these tokens to within a clause.

```
#group AUTHORS: [CC][UL] <Shakespeare>, <Faulkner>, <Hemingway> [/UL] [/CC]
```

The unordered list marker can be thought of as a container much like the clause container marker `[CC]` rather than a marker like the `[CL]` marker. The elements that it matches do not have to be contiguous.

For example, the above rule also matches `Faulkner`, `Amis`, `Shakespeare`, `Wolfe`, `Bukovsky` and our old friend `Hemingway` .

This marker is invalid when:

- It is inside range delimiter `[]`, item iterator `{}`, and token brackets `<>`
- It is found in `#define` statements
- It is inside other unordered list `[UL]` markers
- It is not used in pairs
- It is used more than once in a single rule

3.7.10 Unordered Contiguous List Marker [UC]

Use the unordered contiguous list marker `[UC] expr1, expr2, ..., exprN [/UC]` like the unordered list marker, with the additional restriction that all the listed elements should form a contiguous string.

Note:
This marker impacts processing time; use it sparingly.

Related Topics

- Unordered List Marker [UL]

3.8 Writing Extraction Rules Using Context Markers

You can create more complex rules that allow for discontinuous pattern matching by using multiple sets of context markers (`[OD][/OD]`) in the same rule.

Example

If you run the following rule:

```
#group PERSON_BIRTH: [OD Name] [TE PERSON] < >+ [/TE] [/OD] < >* <STEM:be><born> < >* [OD Date_Birth][TE
DATE|YEAR] < >+[/TE][/OD]
```

against the following input text:

Max Kauffman was born in Breslau on the 11th of December, 1882.

The output is as follows:

```
PERSON_BIRTH: "Max Kauffmann was born in Breslau on the 11th of December, 1882"
Name: Max Kauffman
Date_Birth: 11th of December, 1882
```

You do not have to provide a name (`[OD Name]`) for the different fields in your rule, as follows:

```
#group PERSON_BIRTH: [OD] [TE PERSON] < >+ [/TE] [/OD] < >* <STEM:be><born> < >* [OD][TE DATE] < >+[/TE][/OD]
```

In this case, extraction processing supplies a default name for the subentry categories, as follows:

```
PERSON_BIRTH: "Max Kauffmann was born in Breslau on the 11th of December, 1882"
PERSON_BIRTH-1: Max Kauffman
PERSON_BIRTH-2: 11th of December, 1882
```

However we do recommend that you use your own names for the fields to make your rule more meaningful. The internal structure of the entities found by these rules is similar to those of the extraction processing built-in entities and subentities.

3.9 Regular Expression Operators Supported in CGUL

This section describes all regular expression operators that are valid in CGUL. They are grouped into the following categories:

- standard operators
- iteration operators
- grouping and containment operators
- operator precedence

3.9.1 Standard Operators Valid in CGUL

Standard regular expression operators valid in CGUL are described in the following table.

Operator	Description
Character wildcard (`.`)	Matches any single character.
Alternation (`|`)	Acts as a Boolean OR, which allows the combination of two or more expressions or alternatives in a single expression.

Operator	Description
Escape (`\`)	Escapes special characters so they are treated as literal values.
Character negation (`^`)	Specifies a negated character (the character following the caret `^` symbol).
String negation (`~`)	Specifies a negated string (the string, inside parenthesis, following the `~` symbol).
Subtraction (`-`)	Specifies the subtraction of one expression from the results of another.
Character Classifier (`\p`)	Specifies the character in the character class specified by `{value}`. Currently, two classes are supported: • `\p{ci}`–matches the expression with the input string regardless of case (case insensitive) • `\p{di}`–matches the expression with zero or more of the diacritics specified in the expression (diacritic insensitive)

Related Topics
- Character Wildcard (.)
- Alternation (|)
- Escape (\)
- Character Negation(^)
- String Negation(~)
- Subtraction (-)
- Character Classifier (\p)

3.9.1.1 Character Wildcard (.)

The character wildcard (`.`) operator matches any single character.

In the following example, the expression matches the single character in between the literal characters:

```
#subgroup QAEDA: <Qa.da>
```

This matches **Qaeda** and **Qaida**, as well as combinations such as **Qasda**, **Qavda**, and so on.

Note:
This operator is invalid inside item iterator `{}` and range operator `[]` brackets.

3.9.1.2 Alternation (|)

The alternation operator (`|`) acts as a Boolean OR in regular expressions, which allows the combination of two or more items or alternatives. If any of the alternatives is matched, then the alternation group is treated as a match. Items to be alternated should be between parenthesis.

In the following example, the expression matches any one of the specified characters, in this case, **a**, **b**, **c**, or **d**.

```
a|b|c|d
```

In the following example, the expression matches working and worked.

```
<work(ing|ed)>
```

In the following example, there is more than one token on one side of the alternation. In this case, you should use parenthesis to group the expression that contains multiple tokens.

```
<elm>|(<(ap|ma)ple><tree>)
```

This matches elm, apple tree, and maple tree.

If alternation involves two separate tokens, each token must be within angle brackets `<>`. For example, `<hit, STEM: hit>|<attack.*, POS:V>` is correct.

Note:
The `|` operator is invalid at the beginning of an expression, and within item iterator `{}` and range operator `[]` brackets. The | operator must have two operands.

3.9.1.3 Escape (\)

The escape (`\`) operator escapes special characters that need to be treated as literal characters. The following symbols must be escaped to be used literally:

```
\ : ! ? ( ) . - [ ] { } | * < > + % ~ , ^ @
```

In the following example, the \ operator escapes the following symbols so they can be treated as literal characters: `< > /`

```
#group Title: <\<> <title> <\>> <\<> <\/> <title> <\>>
```

This matches `<title>` and `</title>`.

Note:
Only valid when preceding a special character.

3.9.1.4 Character Negation(^)

The character negation operator (`^`) specifies which character to negate. This causes a match to all characters except the negated one. The negated character follows the caret ^ symbol.

In the following example, the expression matches **bbb**, **bcb**, **bdb**, and so on, but not **bab**.

```
b^ab
```

In the following example, the expression matches **bdb**, **beb**, **bfb**, and so on, but not **bab**, **bbb**, or **bcb**

```
b^(a|b|c)b
```

Note:
Care should be taken when using character negations. For instance, alternating negations will invalidate the entire expression:

```
(^a|^b)
```

The above example will apply to all tokens that are not **a** as well as all tokens that are not **b**: in other words, all tokens.

Note:
Character classifiers do not operate with character negation.

3.9.1.5 String Negation(~)

The string negation operator (`~`) specifies which string to negate. This causes a match to any string except the negated one. The negated string follows the tilda symbol (`~`) and must be contained within parentheses.

In the following example, the expression matches any string except car.

```
~(car)
```

In the following example, the expression matches any token that is not a Noun.

```
POS:~(Nn)
```

In the following example, the expression matches any token that is not a Noun or a verb.

```
POS:~(Nn|V)
```

Note:
Care should be taken when using string negations. For instance, alternating negations will invalidate the entire expression:

```
<POS:~(Adj)|~(Nn)>
```

The above example will apply to all tokens that are not `Adj` as well as all tokens that are not `Nn`: in other words, all tokens.

Note:

- String negation should be used sparingly as it is costly in terms of processing time. Use only when other ways of expressing are unavailable.
- Character classifiers do not operate with string negation.

3.9.1.6 Subtraction (-)

The subtraction operator (`-`) specifies a subset of strings defined by one expression that are not also defined by another. That is, matches from the first expression are only valid when they are not matched by the second.

In the following example, the expression matches all expressions beginning with **house**, except for strings ending in **wife**, such as **housewife**.

```
house.* - .*wife
```

Note:
This operator is only valid between two expressions that do not contain range operators or character classifiers.

3.9.1.7 Character Classifier (\p)

The character classifier operator (`\p {value}`) specifies that the input string matches the character class specified by `{value}`. The possible values are:

- `\p{ci}`–matches the expression with the input string regardless of case (case insensitive)
- `\p{di}`–matches the expression exactly or the expression with an input string that contains a subset of zero or more of the diacritics specified in the expression (diacritics insensitive)

Note:
`\p{di}` operator is used to allow matches on input that either has full correct diacritics, partially correct diacritics, or no diacritics at all.

- For example, consider the following rule and different strings as input to the rule:

  ```
  #group Eleve: <\p{di}( élève)>
  ```

  The rule will match the following four input strings:

  - `élève`

    (correct diacritics)

  - `eleve`

    (no diacritic at all)

  - `elève`

    (correct one only on 2nd “e”)

  - `éleve`

    (correct one only on 1st “e”)

  The rule will not match the following three input strings:

  - `éléve`

    (incorrect diacritic on 2nd “e”)

  - `èléve`

    (incorrect diacritic on both “e”)

  - `elevé`

    (no diacritic on first 2 “**e**”, and incorrect one on 3rd “**e**”)

In the following `{ci}` examples, the expressions match both **Usa** and **usa**.

```
<\p{ci}Usa>
<\p{ci}usa>
```

In the following `{ci}` examples, the expressions match any character case combination, such as **USA**, **Usa**, **usa**, **usA**, and so on.

```
<\p{ci}(Usa)>
<\p{ci}(usa)>
\p{ci}<usa>
\p{ci}<USA>
```

In the following `{ci}` example, the expression consists of a sequence of tokens.

```
\p{ci}(<the><united><states>)
```

In the following `{di}` example, the expression matches **blasé** and **blase**.

```
<blas\p{di}é>
```

In the following `{di}` example, the expression matches **blase** only

```
<blas\p{di}e>
```

In the following `{di}` example, the expression matches **élève**, **elève**, **éleve**, and **eleve**, but not **éléve**.

```
\p{di}<élève>
```

This operator is invalid when

- It is found inside `{}` (iteration braces)
- It is found within the range operator `[]`
- It is used on a range
- In subtraction expressions
- In expressions that contain negation operators
- In tokens that contain `STEM` or `POS` expressions, unless the classifier is assigned to the `STEM` value. However, operations on `POS` values are invalid
- The value is not defined
- It contains wildcards, character classifiers cannot contain wildcards

3.9.2 Iteration Operators Supported in CGUL

In CGUL iteration is expressed by four basic symbols as described in the following table.

Operator	Description
Question Mark (`?`)	Matches zero or one occurrence of the preceding item
Asterisk (`*`)	Matches zero or more occurrences of the preceding item
Plus sign (`+`)	Matches one or more occurrences of the preceding item
Braces (`{}`)	Indicates an item iterator that matches a specific number of occurrences of the preceding item

Note:
Iterators used inside of a token match the item or expression it follows in sequence, without blank spaces. For example, `<(ab){2}>` would match `abab`. Iterators used outside of a token iterate the token it follows, match each token individually. For example, `<(ab)>{2}` would match **ab ab**.

Related Topics

- Question Mark (?)
- Asterisk (*)
- Plus Sign (+)
- Braces ({ })

3.9.2.1 Question Mark (?)

The question mark (`?`) operator is always used following a literal character, special character, or expression grouped as an item. The `?` operator matches zero or one occurrence of the item it follows.

In the following example, the `?` operator matches zero or one occurrence of h, d, and h respectively.

```
#subgroup GADAFY: <(G|Q)adh?d?h?a+f(y|i y?)>
```

This matches **Gadafy**, **Gaddafy**, **Gadafi**, **Gaddafy**, **Qadhafi**, **Qadhdhaafiy**, and so on.

This operator is invalid when:

- It is preceded by `*`, `+`, or `?`
- It is found within the item iterator `{}`
- It is found within the range operator `[]`
- It is found in POS values

3.9.2.2 Asterisk (*)

The asterisk (`*`) operator is always used following a literal character, special character, or expression grouped as an item. The `*` operator matches zero or more occurrences of the item it follows.

In the following example the `*` operator matches zero or more occurrences of an adjective preceding words with the stem **animal**.

```
#subgroup Animals: <POS: Adj>* <STEM:animal>
```

This matches **animal**, **wild animal**, **poor mistreated animals**, and so on.

This operator is invalid when:

- It is preceded by `*`, `+`, or `?`
- It is found within the item iterator `{}`
- It is found within the range operator `[]`
- It is found in `POS` values

3.9.2.3 Plus Sign (+)

The plus sign `(+)` operator is always used following a literal character, special character, or expression grouped as an item. The + operator matches one or more occurrences of the item it follows.

In the following example, the `+` operator matches one or more occurrences of lowercase alphabetic characters that follow an uppercase alphabetic character.

```
#group PROPERNOUNS: <[A-Z][a-z]+>
```

This matches any word that starts with a capital and continues with one or more lowercase letters.

In the following example, the `+` operator matches any ending for words that start with **activat**.

```
#subgroup Active: <activat.+>
```

This matches **activation**, **activate**, **activator**, **activated**, and so on.

This operator is invalid when:

- It is preceded by `*`, `+`, or `?`
- It is found within the item iterator `{}`
- It is found within the range operator `[]`
- It is found in `POS` values

3.9.2.4 Braces ({ })

Braces are used to indicate an item iterator that matches a specific number of occurrences of the expression it follows. This iterator is always used following a literal character, special character, or expression grouped as an item.

You can use this iterator in one of two ways:

- `{m}`–Matches **m** (1 to 9) occurrences of the preceding item
- `{m, n}`–Matches between **m** and **n** (0 to 99) occurrences of the preceding item

Note:
This iterator re-evaluates the expression it follows for each iteration, therefore it looks for subsequent occurrences of the expression.

Example

In the following example, the item iterator matches numbers that contain four digits and a hyphen followed by four more digits.

```
#define ISSN_Number: [0-9]{4}\-[0-9]{4}
```

This matches **2345-6758**.

The use of the iterator causes the extraction process to match four consecutive (contiguous) digits only, if the input contains groups of three digits, or four digits separated by other characters, then there would be no match.

In the following example, the item iterator matches strings that start with a single uppercase or lowercase alphabetic character, followed by zero or one hyphen, followed by three digits between 0 and 6.

```
#define SoundexIndex: [A-Za-z]\-?[0-6]{3}
```

This matches **S543**, **d-563**, but does not match **S54** or **d-5F4D3**.

In the following example, the item iterator matches sentences that are composed by zero or one determiner, zero to three adjectives, and one or more nouns or proper nouns.

```
#group NounPhrase: <POS: Det>?<POS: Adj>{0,3}<POS: Nn|Prop>+
```

This matches Young single white female.

Note:
This iterator is invalid at the beginning of an expression or when found within braces.

3.9.3 Grouping and Containment Operators Supported in CGUL

In CGUL, grouping and containment can be expressed by the operators as described in the following table.

Operator	Description
Range delimiter (-)	Specifies a range of characters when used inside a character class, enclosed in square brackets, such as `[a-z]`.
Range operator (`[]`)	Indicates a character class.

Operator	Description
Item grouper (`()`)	Groups items together so they are treated as a unit.

Related Topics
- Range Delimiter (-)
- Range Operator ([])
- Item Grouper ( )

3.9.3.1 Range Delimiter (-)

The range delimiter (`-`) specifies a range of characters when used inside a character class (meaning, inside the square brackets that represent the range operator). The range is from low to high inclusive.

The following example specifies three character ranges:

```
#define ALPHANUMERIC: [A-Za-z0-9]
```

This matches uppercase or lowercase alphabetic characters or numeric characters.

Note:
The range delimiter is only valid within range operator brackets `[]`.

3.9.3.2 Range Operator ([])

The range operator (`[]`) is used to indicate a character class. A character class is a range of characters that may be used in the current regular expression to match. This character class matches a single character, regardless of how many characters are defined within the character class.

Characters contained within the brackets can be individually listed or specified as a character range using the range delimiter (`-`). A character class can contain as many individual characters as needed and can contain multiple character ranges.

Note:

- Blank spaces within the range operator will cause a syntax error.
- The range operator is invalid inside all brackets except `()` and `<>`.

The following example defines a set of single characters:

```
#define Vowels: [aeiou]
```

This is the same as `(a|e|I|o|u)` and matches any of **a**, **e**, **I**, **o**, and **u**.

The following example denotes a character range, in ascending order.

```
#define ALPHAUPPER: [A-Z]
```

The following example specifies several character ranges:

```
#define ALPHANUMERIC: [A-Za-z0-9]
```

This matches any uppercase or lowercase alphabetic character or any numeric character.

The following example specifies a range of characters and individual characters together.

```
#define ALPHA: [A-Za-záéíóúñ]
```

Note:
No special characters are allowed inside range operators other than the range delimiter (`-`). Also, character classifiers do not operate within range operators (meaning, `[\p]` is interpreted as "match either the backslash or the letter **p**") .

3.9.3.3 Item Grouper ( )

The item grouper `()` is used to group a series of items into a larger unit. The group is then treated as a single unit by CGUL operators.

In this example, the initial character for each word is grouped to enable alternation between the uppercase and lowercase letter. Also, each group of related tokens is grouped to enable alternation with each other.

```
#group AIR_POLLUTANTS: (<(S|s)ulphur> <(D|d)ioxide>) | (<(O|o)xides> <of> <(N|n)itrogen>) | <(L|l)ead>
```

This matches **Sulphur Dioxide**, **sulphur dioxide**, **Sulphur dioxide**, **sulphur Dioxide**, **Oxide of Nitrogen**, **oxide of nitrogen**, **Oxide of nitrogen**, **oxide of Nitrogen**, **Lead**, and **lead.**

3.9.4 Operator Precedence Used in CGUL

CGUL has a precedence hierarchy which determines in which order the operators bind their environment. The following table shows the order from top to bottom.

Functions	Operators
Predefined items	`%`
Escape	`\`
Item iterator (`{}`), token (`<>`) and grouping brackets (`[]`)	`{ }`, `<>` , `[]`
Item grouper	`()`
Character Classifier	`\p`
Negation	`^` , `~`
Subtraction	`-`
Item iterators	`*`, `+`, `?`
Concatenation	(no operator)
Alternation	`|`

3.9.5 Special Characters

CGUL special characters must be escaped when referring to them as part of an expression.

%	\	|	.
,	+	?	!
(	)	[	]

{	}	<	>
^	:	*	~
-	@		

3.10 Match Filters Supported in CGUL

CGUL supports the match filters described in the following table.

Operator	Description
Longest match	By default, only the longest match is returned.
Shortest match `?`	Forces the shortest match on the preceding token expression.
List `*`	Lists all matches without filtering.

Related Topics

- Longest Match Filter
- Shortest Match Filter (?)
- List Filter (*)

3.10.1 Longest Match Filter

The longest match filter does not have to be specified as it is used by default, except inside markers (such as `NP`, `VP`, `TE`, and so on) where the shortest match applies and cannot be overruled. Only the longest match applies to the preceding token expression.

For example:

```
#group ADJNOUN: <POS:Prop>* <>* <POS:Nn>
```

Using the following text:

- Jane said Paul was a baker and Joan was once a carpenter

This example will match

- Jane said Paul was a baker and Joan was once a carpenter.

Note:
It will match because "Jane" is a proper noun (`Prop`) and "carpenter" is a regular noun, so everything in between matches the `<>*` operator.

Related Topics
- Match Filters Supported in CGUL
- Shortest Match Filter (?)
- List Filter (*)

3.10.2 Shortest Match Filter (?)

The shortest match filter forces the shortest match on the preceding token expression.

For example:

```
#group ADJNOUN: <POS:Prop> <>*? <POS:Nn>
```

Using the following text:

- Jane said Paul was a baker and Joan was once a carpenter

This example will match

- Jane said Paul was a baker
- Joan was once a carpenter

The shortest match is calculated from every starting point. However, any match that partly or fully overlaps a previous match is filtered out. As a result, the above example will not match "Paul was a baker" because that is overlapped by the larger match: "Jane said Paul was a baker".

Note:
Valid only when preceded by a token wildcard `<expr>+` or `<expr>*`, where `expr` can be any valid expression or empty.

Related Topics
- Match Filters Supported in CGUL

- Longest Match Filter
- List Filter (*)

3.10.3 List Filter (*)

The list filter returns all matches.

For example:

```
#group ADJNOUN: <POS:Prop> <>** <POS:Nn>
```

Using the following text:

- Jane said Paul was a baker and Joan was once a carpenter

This example will match

- Jane said Paul was a baker
- Paul was a baker
- Jane said Paul was a baker and Joan was once a carpenter
- Paul was a baker and Joan was once a carpenter
- Joan was once a carpenter

Note:
The list filter is valid only when preceded by a token wildcard`<expr>+` or `<expr>*`, where `expr` can be any valid expression or empty.

Related Topics
- Match Filters Supported in CGUL
- Longest Match Filter
- Shortest Match Filter (?)

3.11 Compiling Extraction Rules

Once you create a text file that contains your extraction rules, you use the rule compiler to compile your rules into `.fsm` files and to check the syntax of your rules. The `.fsm` files can contain multiple custom entity definitions. The rule compiler uses the following syntax.

Note:
The rule compiler does not recognize file names that include blanks.

Syntax

```
tf-cgc -i <input_file> [options]
```

where,

`-i <input_file>` specifies the name of the file that contains the extraction rules. This parameter is mandatory.

`[options]` are the following optional parameters:

Parameter	Description
`-e<encoding>`	Specifies the character encoding of the input file. The options are `unicode`, `utf-8`, `utf-16`, `utf_8`, and `utf_16`. If the parameter is not specified, the rule compiler checks for a BOM marker in the file. This marker informs the compiler about the encoding type. If no BOM is found, the default encoding value is ISO-8859-1.
`-o <filename>`	Specifies the name of the output file. The output file must have an extension of `.fsm`. The default name is `<input_file>.fsm`
`-h`, `-help`, `--help`, `-?`	Displays a message outlining the command options.

Example

To run the rule compiler, type the following at the command prompt:

```
tf-cgc -i myrules.txt
```

where,

`myrules.txt` is the name of the file that contains the custom entity rules.

In this example, the compiler assumes the default character encoding for the input file (`ISO-8859-1`), and the output file is `myrules.fsm`.

CGUL Best Practices and Examples

4.1 Best Practices for a Rule Development

Use the following guidelines when developing rules using CGUL:

1. Anything that is repeated belongs in a `#subgroup` statement, or in case of a character expression, in a `#define` statement.

   For example,

   ```
   !example 1
   #subgroup sympton_adj:  <STEM:stiff|painful|aching|burning>
   #subgroup body_part: <STEM:head|knee|chest|ankle|eye|mouth|ear>
   #group symptoms: %( symptom_adj) %( body_part))
   #group vague_symptoms: %( symptom_adj) <STEM: sensation>

   !example 2
   #define neg_prefix: ((de)|(dis)|(un))
   #group negative_action:
   {
   (<STEM: %(neg_prefix).+, POS :V>) |
   (<POS:V> [NP] <>* <STEM: %(neg_prefix).+, POS :Nn> [/NP])
   }
   ```

   Define and subgroup statements have no impact on the performance of a rule file. They are not compiled separately. Instead, their content is copied into their `%()` placeholders before compilation. While posing no restrictions, they offer three considerable advantages:

   - They reduce the risk of typographical errors
   - They allow faster and easier maintenance
   - They allow reuse in other rules

2. If you use a complicated character expression, assign it to a `#define` statement. If you use a long list of strings, such as alternate stems for one token, assign it to a `#subgroup` statement.

   For example,

   ```
   !example 1
   #define chemical_prefix:
   {
   ((mono)|
   (bi)|
   (di)|
   (tri)|
   (tetra)|
   (octo)|
   (deca)|
   (hepto)|
   (desoxy)|
   (mero)|
   (meso)|
   (ribo))
   ```

```
}

!example 2
#subgroup weather_phenomena:
{
(<STEM:snow> |
<STEM:frost,POS:Nn> |
<STEM:cold,POS:Nn> |
<STEM:rain,POS:Nn> |
<STEM:hail,POS:Nn> |
<STEM:flood(ing)?,POS:Nn> |
<STEM:hurricane> |
<STEM:drought> |
<STEM:tornado> |
<STEM:lightning> |
<STEM:weather,POS:Nn>)
}
```

Even if you intend to use a list only once, storing them in a `#define` or `#subgroup` statement makes your group rules easier to read.

3. Save your reusable `#define` statements and `#subgroup` statements in a separate file. You can include them wherever you need them by using the `#include` statement.

   For example, suppose you have created a rule file called `MyShortcuts.txt`, containing the following statements:

   ```
   #subgroup PERSON: [TE PERSON]<>+[/TE]
   #subgroup NP: [NP]<>+[/NP]
   ```

   You can refer to these expressions in another rule file by including them:

   ```
   #include "MyShortcuts.txt"
   !(...)
   #group purchase: %(PERSON) <STEM:buy> %(NP)
   ```

4. You must declare `#subgroup`, `#define,` and `#include` statements before they are invoked in a rule.

   For example,

   ```
   #subgroup search_expr: <STEM:look><STEM:for>
   #group missing: %(search_expr) [OD missing_obj][NP]<>+[NP][/OD]
   ```

5. The `#subgroup` statement helps make your rule files easier to understand, edit, and test. However, `#subgroup` statements do not reduce the processing speed of the rules. They can be cascaded, though not recursively. Test your `#subgroup` statements by temporarily casting it as a `#group` and running it over a sample text with some entities it is intended to match.

   For example,

   ```
   #subgroup PERSON: [TE PERSON]<>+[/TE]
   #subgroup PGROUP: %(PERSON) (<\,> %(PERSON))* (<and> %(PERSON))?
   #group missing:  <STEM:look><STEM:for> [OD missing_ps]%(PGROUP)[/OD]
   ```

6. Give descriptive names to your `#group`, `#subgroup` and `#define` statements, and output delimiters `[OD name]`. This will make your rules much easier to understand and edit.

   For Example,

   ```
   #subgroup PERSON: [TE PERSON]<>+[/TE]
   #subgroup attack_verb: <STEM:attack|bomb|highjack|kill>
   #group terrorist_attack: [OD terrorist] %(PERSON) [/OD] <>* %( attack_verb)
                                   <>* [OD target] [NP] <>+ [/NP] [/OD]
   ```

7. CGUL is case-sensitive. To match both cases, list the alternatives, as in `<(A|a)ward>`.

For example,

```
#group POLLUTANTS: (<(S|s)ulphur>  <(D|d)ioxide>) | (<(O|o)xides>
                              <of> <(N|n)itrogen>) | <(L|l)ead>
```

CGUL expressions or sub-expressions can be made case insensitive by deploying the character classifier `\p{ci}`. This operator allows matching of strings that differ from the string in the rule only in its case value. For example, `\p{ci}(usa)` will match "USA", "Usa" and so on.

8. The character classifier `\p{di}` forces diacritics insensitivity. This operator allows matching of strings that differ from the string in the rule only in the presence of diacritics. For example, `\p{di}(élève)` matches "elève", "eleve" and so on. For more information, read Character Classifier (\p) before using this character classifier.
9. If the pattern to be matched is non-contiguous, you need to insert a token wildcard, such as `<>*` or `<>+`, in your rule. Be aware that token wildcards affect performance, and that their total number per rule, including those present in invoked subgroups, should be kept within reasonable limits.
10. Try to avoid unbounded token wildcards, such as at the beginning or end of an expression. They lengthen the runtime considerably. Use sentence delimiters (`[SN]`) or punctuation tokens to limit the scope of your wildcards.

    For example,

```
#subgroup say_expr: <STEM:say|remark|conclude|insinuate|insist>
#group quote_sent: [SN] <>*  [TE PERSON]<>+[/TE]  %(say_expr) <>* [\SN]
```

11. The default for token wildcards is the longest match. Match filters are available to force shortest (or all) matches. Tempting as they may be, use them sparingly. When several are invoked within the same rule, matching filters inevitably interact with each other, which can lead to unintended results.

    For example,

```
#group visit: [CC] [OD visitor] [TE PERSON] <> +[/TE] [/OD] <>* <STEM:visit>
                  <>*? [OD visitee] [TE PERSON] <>+ [/TE] [/OD]  [/CC]
```

12. To restrict the processing of a rule to a portion of the input document, you can define an input filter in the group header.

    For example, a rule `myrule` with the following header:

```
#group myrule (paragraph="[1-4]"): ...
```

    is applied only to the first four paragraphs of the input text.

13. To restrict the processing of a rule to a particular sentence in the input document, you can use the scope key and the `[P]` and `[SN]` delimiters to mark the desired sentence.

    For example, to extract the last sentence of the first paragraph:

```
#group myrule (scope="paragraph" paragraph="[1]"): [SN] <>+ [/SN] [/P]
```

14. When clause boundaries (`[CL]`, `[CC]`) are available for the language on which you are working, always consider using them for discontinuous patterns. Matches on such patterns that cross clause boundaries are often unintended.

    For example,

```
#group quote: <STEM:say>[OD quotation][CL]<>+[/CL][/OD]
```

15. Become familiar with the set of parts of speech and the list of entities available for the language on which you are working.

    For more information, refer to the *Text Data Processing Language Reference Guide*.

16. The English language module supports advanced parsing as the default for custom rule-driven extraction. Advanced parsing features richer noun phrases, co-reference analysis of personal pronouns, and syntactic function assignment.

    For an introduction to this parsing method, see the English section of the *Text Data Processing Language Reference Guide*.

    Note:
    Advanced parsing is currently supported for English only.

17. The English module supports two types of noun phrase extraction: one resulting from standard parsing, the other from advanced parsing. The advanced parser returns the noun phrase with its pronouns, numbers and determiners. It also supports noun phrase coordination.

    For example, the following text shows the standard phrases in bold and their advanced phrases underlined:

    Note:
    To process using standard noun phrases, turn advanced parsing off. Since this is a runtime option, you need to take care to group your CGUL rules according to the noun phrase parsing level they require.

18. When you use `[NP]` or `[VP]` markers in CGUL English (advanced parsing mode), consider specifying their syntactic function. This might drastically improve the precision of your rules.

    For more information, refer to the *Text Data Processing Language Reference Guide*.

19. If you want to include a list of lexical entries in a CGUL rule file, there is no need to copy and format them into that rule file. Provided that your list consists of single words separated by hard returns, you can include the list using the following syntax:

    ```
    @lexicon PASSWORDS mypasswords.txt
    ```

    The list can then be referred to as `%(PASSWORDS)`, representing all the entries contained in `mypasswords.txt` as a list of alternate forms. The expression can be used to specify a token string such as `<%(PASSWORDS)>` or a stem, such as `<STEM: %(PASSWORDS)>`.

20. Be careful with entity filters when running the extraction process with extraction rules. If your rules refer to entity types, the entities for these types will have to be loaded, regardless of the filter you have set. As a consequence, the output for certain filters may be different from expected.

4.2 Syntax Errors to Look For When Compiling Rules

This section specifies some common syntax errors that are untraceable by the compiler.

1. Incorrect reference to `#define` and `#subgroup` directives is a very common error that the compiler often cannot detect. For example:

   ```
   #define SZ: (s|z)
   #group SpanishName: <.*gueSZ>
   ```

   This looks for strings ending on `gueSZ` rather than the intended `gues` and `guez`. The correct syntax is `<.*gue%(SZ)>`
2. By default, surround alternations with parentheses ().
3. Be careful with wildcard expressions in `[UL]` unordered lists. Using wildcards expressions that introduces optionality in the list leads to unintended results.

   For example:

   ```
   ([UL]<Fred>,<F.+>*[/UL])
   ```

   unintentionally matches the single item `Fred`. It is the optionality of `*` that results in an unintended single item extraction `Fred`.

4.3 Examples For Writing Extraction Rules

To write extraction rules (also refereed to as CGUL rules), you must first figure out the patterns that match the type of information you are looking to extract. The examples that follow should help you get started writing extraction rules.

Related Topics

- Example: Writing a simple CGUL rule: Hello World
- Example: Extracting Names Starting with Z
- Example: Extracting Names of Persons and Awards they Won

4.3.1 Example: Writing a simple CGUL rule: Hello World

To write a rule that extracts the string Hello World, you create a token for each word, and define them as a group:

```
#group BASIC_STRING: <Hello> <World>
```

where,

`BASIC_STRING` is the group name; names are always followed by a colon (`:`).

`<Hello>` is a token formed by a literal string.

`<World>` is a token formed by a literal string.

If you want to extract the words with or without initial caps, then you would re-write the rule as follows:

```
#group BASIC_STRING: <(H|h)ello> <(W|w)orld>
```

This would match Hello World, hello world, Hello world, and hello World.

The statement uses the alternation operator (|) to match either upper case or lower case initial characters. The parentheses groups the alternating characters together.

4.3.2 Example: Extracting Names Starting with Z

```
!This rule extracts title and last name for persons whose last name
 !starts with Z

#define LoLetter: [a-z]

#subgroup ZName: <Z%(LoLetter)+>
#subgroup NamePart: <(van|von|de|den|der)>
#subgroup Title: <(Mr\.|Sr\.|Mrs\.|Ms\.|Dr\.)>

#group ZPerson: %(Title) %(NamePart){0,2} %(ZName)
```

Lines 1 and 2 are a comment. The exclamation mark (!) causes the compiler to ignore the text that follows it on the same line.

```
!This rule extracts people's full name for persons whose last name
    !starts with Z
```

Line 3 defines a character class for all lowercase alphabetic characters. Character classes are enclosed within the range operator (`[]`), and can use the hyphen (`-`) symbol to indicate a range. Character classes can also include lists of individual characters without indicating a range, or a combination of both, for example `[a-zãæëïõü]`.

```
#define LoLetter: [a-z]
```

Once an item, such as `LoLetter`, is given a name and defined, it can be used within another item, such as a group or subgroup. The syntax is: `%(item)`. The `%` and parentheses are required.

Lines 4, 5, and 6 are subgroups that define tokens that match the different components of a name.

```
#subgroup ZName: <Z%(LoLetter)+>
#subgroup NamePart: <(van|von|de|den|der)>
#subgroup Title: <(Mr\.|Sr\.|Mrs\.|Ms\.|Dr\.)>
```

- `ZName`–Defines a token that matches words that start with the uppercase letter `Z`.

  The token consists of the literal character capital Z, the character class LoLetter, followed by the + iterator. The + iterator matches the item it follows one or more times. Finally, the entire expression is delimited by angle brackets (<>), defining it as a token.

  This rule matches all words that start with an initial capital Z, followed by one or more lower case characters (LoLetter).

- `NamePart`–Defines a token that matches the possible name parts that can be part of a last name.

The token consists of a group of alternating terms. Each term is separated by the alternation operator (`|`), denoting that the expression matches any of the terms within the group of alternating terms. The terms are enclosed within parentheses `()`. The parentheses group the items into a larger unit, which is then treated as a single unit. Finally, the entire expressions is delimited by angle brackets (<>), defining it as a token.

Another way of expressing the same operation would be:

```
#subgroup NamePart: <van>|<von>|<de>|<den>|<der>
```

This rule matches the literal strings van, von, de, den, and der.

- `Title`–Defines a token that matches the possible titles a person can have.

  The token consists of the same components as `NamePart`, with the exception of the literal string content of the token, and the use of the escape symbol (\) to add a period at the end of each literal string. The escape symbol (\) is used to escape special characters so they are handled as literal characters. In this case, if the period is not escaped, it is interpreted as a wildcard.

  This rule matches the literal strings Mr., Sr., Mrs., Ms., Dr.

Line 7 defines the actual entity type for a person's full name, whose last name starts with `Z`.

```
#group ZPerson: %(Title) %(NamePart){0,2} %(ZName)
```

The group consists of a string of the items you defined in the previous lines as subgroups or character classes.

- `Title`–token representing the person's title
- `NamePart`–token representing the person's name parts, if any
- `Zname`–token representing the person's last name

Tokens cannot contain other tokens, therefore, it would be an error to express `%(Title)`, `%(NamePart)`, or `%(Zname)` between angle brackets (<>) within the `ZPerson` group. Items that are not already defined as tokens, must be declared as tokens.

The `{0,2}` iterator that follows the `NamePart` token matches the `NamePart` token zero to two times, with white space between multiple items.

The rule matches names such as Dr. van der Zeller.

4.3.3 Example: Extracting Names of Persons and Awards they Won

```
!Rules for extraction of Award relations

#subgroup Person: [OD Honoree] [TE PERSON] <>+ [/TE] [/OD]
#subgroup Award1: [NP] <~(Academy)>+ <(A|a)ward|(P|p)rize> [/NP]
#subgroup Award2: [UC] <Best|Top> , <POS:Num> [/UC] <POS:Prop>+
#subgroup AnyAward: [OD Award] (%(Award1) | %(Award2)) [/OD]

#group Win: [CC] %(Person) <>*? (<STEM: win|receive|accept> |<named|listed|awarded>) <>*? %(AnyAward) [/CC]
```

Line 1 is a comment. The exclamation mark (`!`) causes the rule compiler to ignore the text that follows it on the same line.

Lines 2, 3, 4 and 5 are subgroups that define tokens that match different components of the rule.

- `Person`–Defines a `PERSON` entity with output delimiters (`[OD Honoree]`).
- `Award1`–Defines a noun phrase that ends in variations of the words **award** and **prize**, but does not include the word **Academy**.
- `Award2`–Defines an unordered, contiguous list that includes combinations of either **Best** or **Top** with a number; followed by one or more proper nouns. For example, **Top Ten Entertainers**, **5 Best Footballers**, and so on.
- `AnyAward`–Defines a subgroup for **Award1** and **Award2** with output delimiters.

Line 6 defines the actual relationship between a **person** and an **award**.

```
#group Win: [CC] %(Person) <>*? (<STEM: win|receive|accept>|<named|listed|awarded>) <>*? %(AnyAward) [/CC]
```

The group consists of a string of the items previously defined as well as intervening words denoting the relation.

The group uses a Clause Container `([CC])` that utilizes the shortest match filter (`?`) to relate a **person** to an **award**.

Testing Dictionaries and Extraction Rules

To test dictionaries and extraction rules, use the Entity Extraction transform in the Designer.

Related Topics

- Designer Guide: Using extraction options

Index