PUBLIC

SAP Data Services

Document Version: 4.2 Support Package 12 (14.2.12.0) – 2020-02-06

Text Data Processing Extraction Customization Guide

THE BEST RUN

Content

1 Text Data Processing Extraction Customization Guide

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. This guide is written for dictionary and extraction rule writers.

2 Introduction

Welcome to the *Extraction Customization Guide*.

The SAP text analytics engine provides a suite of analyzers that enable developers to construct applications for analyzing unstructured data. The analyzers let your application perform extraction and various forms of natural-language processing on unstructured text.

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.

2.1 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 are an application developer or consultant working on enhancing text analysis extraction.
- You understand your organization's text analysis extraction needs.

2.2 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.

3 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. For example, it can store entities composed of common words that have special meaning in your application's domain or it can store alphanumeric sequences such as specialized vocabulary or part numbers. 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.

A dictionary whose name does not specify a language 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:

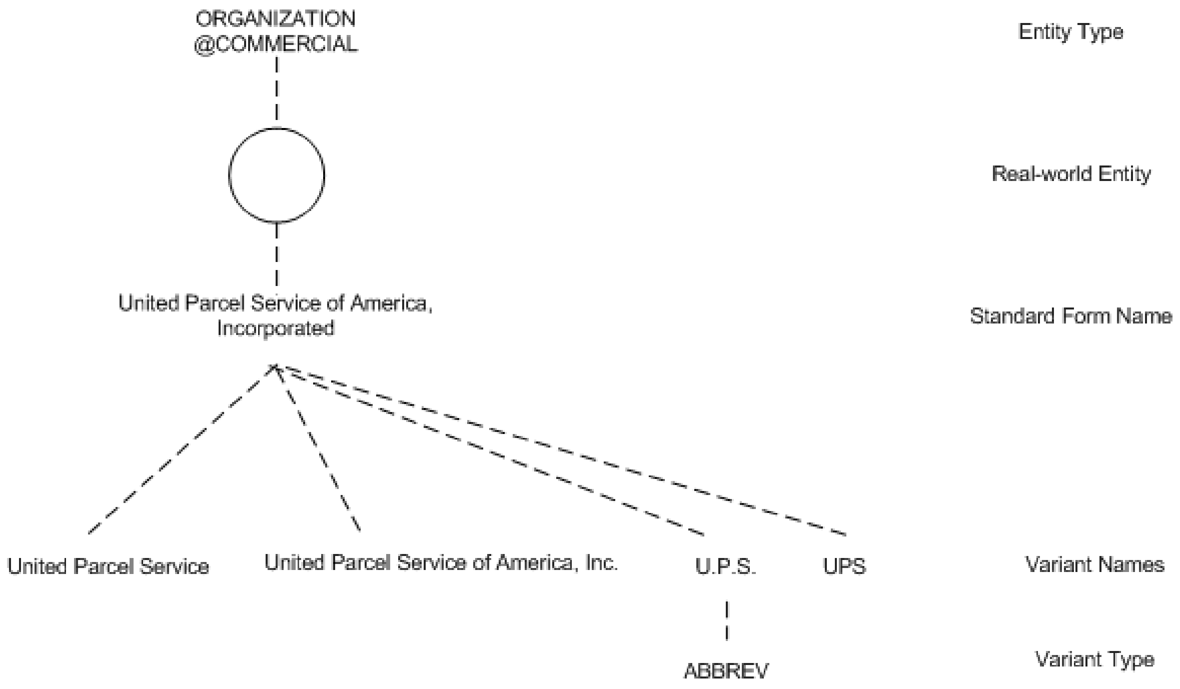
- name variation management, for example, recognizing **Bimmer** as a colloquial reference to the car make **BMW**.
- disambiguation of unknown entities, for example, extracting **DF-21D**, **DF 21D**, **DF21D**, and **df-21d** as alternative forms of the **Dong Feng 21D** rocket.
- control over entity recognition, for example, to force Text Analysis to extract **Dong Feng 21D** as an entity of type WEAPON instead of extracting **Dong Feng** as a PERSON.

3.1 Entity Structure in Dictionaries

A dictionary contains a number of user-defined entity types, each of which contains any number of entities, of standard and variant types.

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. You can also define an arbitrary string to a standard form using the `uid` attribute.
- 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, you may define an arbitrary string to be associated with a variant via its `type` attribute. For example, you might use the string `ABBREV` to mark variants that are an abbreviation of the standard form.
  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 access the standard form and the variant names given any of the related forms.

3.1.1 Generating Predictable (Standard) Variants

Standard variant generation, performed by the dictionary compiler, produces a variety of common synonyms.

The variants **United Parcel Service** and **United Parcel Service of America, Inc.** are common and predictable, and other predictable variants can be generated by the dictionary compiler for later use in the extraction process. The dictionary compiler, using its variant generation 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, if the standard form ends with the abbreviation **Inc.**, **Inc.** will be replaced with **Incorporated** in one of the generated variants, and if the standard form ends with **Incorporated**, **Incorporated** will be replaced with **Inc.** in one of the generated variants.
- Handles optional commas and periods, that is, the optional comma between a company name and the designator (**SAP Labs, Incorporated** vs. **SAP Labs Incorporated**) and the optional period after an abbreviated designator (**SAP Labs Inc.**vs. **SAP Labs Inc**).
- 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 such as the variant **Microsoft** for the standard form **Microsoft Corporation** are not automatically generated, because they are easily misidentified. One-word variants must be entered into the dictionary explicitly. If the standard form does not end with one of the recognized organization designators such as **Inc**, **Corp.**, **Corporation**, and so on, variants will not be generated, even if you specify that the entity should use standard variant generation.

> i Note
>
> Standard variant generation is supported in Dutch, English, French, German, Italian, Portuguese, and Spanish.

3.1.2 Custom Variant Types

Custom variant types let you create your own patterns for generating variants from a standard form.

You define custom variant types in a dictionary. You can specify which tokens from the standard form are included in each variant and then insert different tokens into the variants.

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`.

For another example, you can create a custom variant type that will, for any two-word entity whose second word is **forces** (such as, **Afghan forces**), create variants where **forces** is replaced with **troops**, **soldiers**, **Army**, and **military** (such as, **Afghan troops**, **Afghan soldiers**, **Afghan Army**, and **Afghan military**).

3.1.3 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 type (category) identifier, as in `VEG@STARCHY`. Subtyping is only one-level deep, so `TYPE@SUBTYPE@SUBTYPE` is not valid.

3.1.4 Variant Types

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`.

3.1.5 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 are restricted to match against only part of an input buffer: they are limited to the scope of a sentence. This means that a pattern `"Company *"` will match to the end of the sentence.

i 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 cautiously, keeping potential performance degradations in mind.

3.1.5.1 Wildcard Definitions

The wildcard characters, `*` and `?`, match zero or more tokens within a sentence or exactly one token within a sentence, respectively.

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

3.1.5.2 Wildcard Usage

You can use wildcard characters 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 containing **University**, from the beginning of the sentence to the last occurrence of **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, "I once attended Stanford University, though I considered Carnegie Mellon University" would be returned, and **Stanford University** 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.

3.2 Creating a Dictionary

Create an XML file and compile it.

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 Information

Dictionary XSD [page 10]
Preparing a Dictionary for Use [page 19]

3.3 Extraction Dictionary Syntax

3.3.1 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 (c) 2010-2011, 2014 SAP AG. All rights reserved.
 *
 *  No part of this publication may be reproduced or transmitted in any form or
 *  for any purpose without the express permission of SAP AG. The information
 *  contained herein may be changed without prior notice.
-->
<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:attribute name="transient" type="xsd:boolean"/>
  </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 minOccurs="0" maxOccurs="unbounded">
    <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.	
	`xmlns`	The dictionary's name space. The value must be `"http://www.sap.com/ta/4.0"`.
`entity_category`	The category (type) to which all embedded entities belong. Contains one or more embedded `entity_name` elements.	
	`name`	The name of the category, such as **PEOPLE**, **COMPANY**, **PHONE NUMBER**, and so on. Note that the entity category name is case sensitive.
`entity_name`	A named entity in the dictionary. Contains zero or more of the elements `variant`, `query_only` and `variant_generation`.	
	`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 useful to dictionary editors (it is not used in text analysis).
`variant`	A variant name for the entity. The variant name must be unique within the `entity_name`.	
	`name`	[Required] The name of the variant.
	`type`	[Optional] The type of variant, the value of which may be any arbitrary string. This attribute is useful only to dictionary editors (it is not used in text analysis).

Element	Attributes and Description	
`query_only`	A query_only variant. Query-only variants are not used during the text analysis process. A dictionary editor may mark a variant as query_only to keep it in the dictionary but prevent it from being used during extraction.	
	`name`	[Required] The name of the query-only variant.
	`type`	[Optional] The type of variant, generally is a subtype of the larger `entity_category`.
`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.	
	`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 construction of variants.
`define-variant_generation`	Defines a custom variant generation pattern whose name can be specified in the type attribute of `variant_generation` elements.	
	`name`	Unique name of the custom variant generation pattern.
`pattern`	A wildcard expression for matching `standard_form` or `base_text` values. May contain one or more capture groups. A `define-variant_generation` element may contain multiple `pattern` elements.	
	`name`	The pattern's wildcard expression.
`generate`	Associated with a `pattern` element, `generate` specifies how to construct a variant using the pattern's wildcard expression. Contains one or more placeholders that correspond to capture groups in the pattern.	
	`name`	An expression containing placeholders and static tokens, used to construct a variant form from a `standard_form` or `base_text`.

Related Information

Dictionary Source Format [page 15]

3.3.2 Guidelines for Naming Entities

Provides several guidelines for the format of standard form and variant names in a dictionary.

- Use only characters that are valid in 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;`

3.3.3 Character Encoding in a Dictionary

A dictionary supports most popular character encodings, including ASCII, UTF-8, UTF-16, ISO-8859-1, and so on. 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, 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.

i Note

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

3.3.4 Dictionary Sample File

Here is a brief sample dictionary file.

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <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 Information

Entity Structure in Dictionaries [page 6]

3.3.5 Dictionary Source Format

The dictionary source follows the XSD and can include entries for entity type, entity, entity subtype, variants and variant type.

Example: Entity Type

This example dictionary defines the entity type (category) `PART_NUMBER` and a few entities of that type:

```
<?xml version="1.0" encoding="utf-8"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <entity_category name="PART_NUMBER">
        <entity_name standard_form="81-117"/>
        <entity_name standard_form="81-147"/>
        <entity_name standard_form="840-3-21"/>
        <entity_name standard_form="01-90-3433"/>
        <entity_name standard_form="01-90-8140"/>
        <entity_name standard_form="01-90-3432"/>
    </entity_category>
</dictionary>
```

Example: Entity

This example dictionary adds two new entities to the built-in (out-of-the-box) `ORGANIZATION@COMMERCIAL` category and specifies some variant forms for each entry.

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <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>
```

You may include the optional `uid` attribute for an `entity_name` element. In the example above, you could change the 4th line to

```
<entity_name standard_form="Seventh Generation Incorporated" uid="Company
024641309">
```

where **Company 024641309** is an arbitrary string associated with this entity, such as a unique record identifier into a database containing additional information about the entity. The `uid` attribute value is not used in the extraction process or returned in results of text analysis.

Example: Entity Subtype

This example refines the Entity Type example, creating two subcategories of part numbers:

```
<?xml version="1.0" encoding="utf-8"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <entity_category name="PART_NUMBER@AUTOPSY_SAW">
        <entity_name standard_form="81-117"/>
        <entity_name standard_form="81-147"/>
        <entity_name standard_form="840-3-21"/>
    </entity_category>
    <entity_category name="PART_NUMBER@THERMOTIC_PUMP">
        <entity_name standard_form="01-90-3433"/>
        <entity_name standard_form="01-90-8140"/>
        <entity_name standard_form="01-90-3432"/>
    </entity_category>
</dictionary>
```

Example: Variants and Variant Types

This example extends the dictionary from the preceding section, specifying variant forms of each part number. The standard form represents the OEM part number, the variants include a distributor's inventory ID and the name of the part:

```
<?xml version="1.0" encoding="utf-8"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <entity_category name="PART_NUMBER@AUTOPSY_SAW">
        <entity_name standard_form="81-117"/>
            <variant name="STB003"/>
            <variant name="front bearing"/>
        </entity_name>
        <entity_name standard_form="81-147"/>
            <variant name="STB004"/>
            <variant name="needle bearing"/>
        </entity_name>
        <entity_name standard_form="840-3-21"/>
            <variant name="STS018"/>
            <variant name="switch assembly"/>
        </entity_name>
    </entity_category>
    <entity_category name="PART_NUMBER@THERMOTIC_PUMP">
        <entity_name standard_form="01-90-3433"/>
            <variant name="GOV011"/>
            <variant name="suction valve (inlet)"/>
        </entity_name>
        <entity_name standard_form="01-90-8140"/>
            <variant name="GOV013"/>
            <variant name="aerovent rubber valve"/>
        </entity_name>
        <entity_name standard_form="01-90-3432"/>
            <variant name="GOV012"/>
            <variant name="pressure valve (outlet)"/>
        </entity_name>
    </entity_category>
</dictionary>
```

The optional `type` attribute of the `variant` element may be used to store additional information about a variant. For example, the preceding example could use the `type` attribute to designate which variant is a distributor's inventory ID and which is a part name:

```
    <entity_name standard_form="81-117"/>
        <variant name="STB003" type="Inventory ID"/>
        <variant name="front bearing" type="Part Name"/>
    </entity_name>
    <entity_name standard_form="81-147"/>
        <variant name="STB004" type="Inventory ID"/>
        <variant name="needle bearing" type="Part Name"/>
    </entity_name>
    . . .
```

The `type` attribute can be useful to dictionary editors for organizing dictionary content, but it is neither used during text analysis nor returned in the results.

Example: Standard Variant Generation

To generate standard variant types, include a `variant_generation` tag in an entity's definition.

The dictionary below specifies that the variants of the standard form **Seventh Generation Inc** should be generated using the standard variant generation pattern for English.

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <entity_category name="ORGANIZATION@COMMERCIAL">
        <entity_name standard_form="Seventh Generation Inc">
            <variant_generation type="standard" language="english" />
        </entity_name>
    </entity_category>
</dictionary>
```

The standard variant generation pattern is described in Generating Predictable (Standard) Variants [page 7].

In the `language` attribute, specify the language of the variant generation to apply; standard variant generations are language dependent. Variant generation is supported in Dutch, English, French, German, Italian, Portuguese, and Spanish. If the language attribute is not specified, the standard English pattern is used.

The optional `base_text` attribute of `variant_generation` specifies text different from the standard form to be used when generating variants. In the example above, if you added `base_text="7th Generation Inc"` to the variant_generation element, the variants would include **7th Generation Inc** instead of **Seventh Generation Incorporated**, **7th Generation Corp** instead of **Seventh Generation Corp**, and so on. The `base_text` value must end with one of the standard organizational designators or else variants will not be generated.

Example: Custom Variant Generation

To generate custom variant types, define a name with the list of variant generations. In the following example, the dictionary defines the custom variant type `VariantPattern_Troops` and specifies that the variants for the entity **Afghan forces** and **Afghani forces** are to be generated using that variant type.

```
<?xml version="1.0" encoding="utf-8"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <define-variant_generation type="VariantPattern_Troops" >
        <pattern string="(?) forces" >
            <generate string="\1 troops" />
            <generate string="\1 soldiers" />
            <generate string="\1 Army" />
            <generate string="\1 military" />
        </pattern>
    </define-variant_generation>
<entity_category name="FORCES">
    <entity_name standard_form="Afghan forces">
        <!--  to generate variants for "Afghan forces" -->
        <variant_generation type="VariantPattern_Troops"/>
        <!--  to generate variants for "Afghani forces" -->
        <variant_generation type="VariantPattern_Troops" base_text="Afghani
forces" />
    </entity_name>
</entity_category>
</dictionary>
```

The pattern above uses the `?` wildcard character that matches a single token. Patterns can also contain the `*` (Kleene star) wildcard character that matches zero or more tokens. Surrounding a wildcard with parentheses creates a capture group, which makes patterns more useful by allowing them to perform string substitution via placeholders.

In this example, `(?)` is the capture group and `\1` is the placeholder. This pattern matches entity names whose standard form is the word **forces** preceded by a single token. Thus, it matches **Afghan forces** but not **South Sudan forces** or **French forces royale**.

As this pattern generates variant forms, it replaces `\1` with the string matching the capture group. Thus, for the standard form **Afghan forces** it generates the variants **Afghan troops**, **Afghan soldiers**, **Afghan Army**, and so on.

If an entity's `variant_generation` type is a custom variant having no patterns that match the entity's standard form (or the `variant_generation` includes the optional `base_text` attribute and no patterns match the `base_text` value), no variants will be generated.

Patterns containing more than one capture group can have multiple placeholders in their `generate` tags. Placeholders are numbered in accordance with the left-to-right order in the pattern of their corresponding capture group. For example, if a pattern string contains `(?) forces (?)` and one of its generated strings contains `\1 troops \2`, `\1` will be replaced with the text matching the `(?)` to the left of **forces** and `\2` will be replaced with the match of `(?)` to the right. That is, the standard form **French forces royale** would have the generated variant **French troops royale**.

Using this dictionary will result in the extraction all of the following strings as an entity of type FORCES:

Afghan forces, **Afghan troops**, **Afghan soldiers**, **Afghan Army**, **Afghan military**

Afghani forces, **Afghani troops**, **Afghani soldiers**, **Afghani Army**, **Afghani military**

Example: Wildcard Variant

In this dictionary, the entity has a variant whose name contains the Kleene star ("*") wildcard character:

```
<?xml version="1.0" encoding="windows-1252"?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
    <entity_category name="BANKS">
        <entity_name standard_form="American banks">
            <variant name="Bank of * America" />
        </entity_name>
    </entity_category>
    ...
</dictionary>
```

This wildcard entry means that entities like **Bank of America**, **Bank of Central America**, **Bank of North and South America**, and so on will be matched as variants of **American banks**.

Variant names can also contain the `?` wildcard character. If `?` were used in this example instead of `*`, **Bank of Central America** would still be a variant because it contains exactly 1 token between **Bank of** and **America**, but not **Bank of America** (because there are 0 tokens between **Bank of** and **America**) or **Bank of North and South America** (because there are 3 tokens between **Bank of** and **America**).

3.4 Preparing a Dictionary for Use

When your source file is complete, you must compile it before using it in the text extraction process.

Create an XML file containing your content, formatted according to the dictionary syntax.

> **i 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 Information

Dictionary XSD [page 10]

3.4.1 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 shown in the table below.

`<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. The default location for the language modules is the current working 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. If the dictionary should be used only when processing documents in a specific language, the output filename must begin with the name of that language. For example, if the dictionary is meant to be used only with German documents, the compiled dictionary name must start with "german", and if it is meant to be used only when processing Japanese documents, the output filename must start with "japanese". If the output file does not start with a language name, it will be used regardless of the input document's language.
`-v`	Indicates verbose. Shows progress messages.

Syntax	Description
`-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. ℹ 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's name is `tf.nc-config`. Its default location is the directory specified in the `-d` option. If the `-d` option is not specified, the default location is the current working directory.
`-case_sensitive`	Generates case-sensitive variants. ℹ Note If you include this option, then for each entity in your dictionary that can appear capitalized in multiple ways, you must create a variant capitalized each way the entity can be capitalized.
`-case_insensitive`	Generates case-insensitive variants. ℹ 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.

ℹ 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 Information

Dictionary XSD [page 10]

3.5 Customization of Sentiment Analysis Dictionaries

Sentiment Analysis is customizable via a “thesaurus” dictionary. Users are able to add or remove sentiment keywords as well as change the category of a keyword (for example, from WeakNegative to MinorProblem).

These keywords are in turn called by rules that are syntactically aware. Depending on the domain your application is aimed at, sentiment keywords can be added to improve recall or removed to avoid over-generation.

Customizable Sentiment Analysis is available in the following languages:

- English
- Chinese (Simplified)
- Chinese (Traditional)
- French
- German
- Italian
- Portuguese
- Russian
- Spanish

Each language module has its own Sentiment Analysis “thesaurus” dictionary. If your application is needed in various languages for the same domain, each dictionary will need to be edited to match your needs.

Related Information

Sentiment Analysis Fact Extraction

3.5.1 Whitespace Language Sentiment Analysis Customization

The customizable dictionary is in XML format. Sentiment keywords are grouped in entity categories by sentiment type (weak positive, strong positive, weak negative, and so on) and part-of-speech tag.

Format of Customizable Dictionary

The following shows an excerpt of the formatting and keyword grouping for the Major Problem category for the English dictionary:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dictionary xmlns="http://www.sap.com/ta/4.0">
<entity_category name="MAP@Adj">
<entity_name standard_form="abusive" />
<entity_name standard_form="blurriest" />
<entity_name standard_form="botched" />
<entity_name standard_form="brainless" />
<entity_name standard_form="broken" />
<entity_name standard_form="buggiest" />
<entity_name standard_form="bulkiest" />
</entity_category>
<entity_category name="MAP@Adv">
<entity_name standard_form="botchedly" />
<entity_name standard_form="chaotically" />
<entity_name standard_form="corruptedly" />
<entity_name standard_form="defectively" />
<entity_name standard_form="destructively" />
</entity_category>
<entity_category name="MAP@Noun">
<entity_name standard_form="a real hard time" />
<entity_name standard_form="a really hard time" />
<entity_name standard_form="attrition" />
<entity_name standard_form="attritions" />
<entity_name standard_form="chaos" />
</entity_category>
<entity_category name="MAP@Verb">
<entity_name standard_form="abort" />
<entity_name standard_form="aborted" />
<entity_name standard_form="aborting" />
<entity_name standard_form="aborts" />
<entity_name standard_form="botch" />
<entity_name standard_form="botched" />
<entity_name standard_form="botches" />
<entity_name standard_form="botching" />
<entity_name standard_form="broken" />
</entity_category>
</dictionary>
```

The names of the entity categories in the dictionary generally follow the pattern “Output Value” + “@” + “part of speech”:

Category	Meaning
MAP@Adj	MajorProblem adjectives

Category	Meaning
MAP@Adv	MajorProblem adverbs
MAP@Noun	MajorProblem nouns
MAP@Verb	MajorProblem verbs

This table shows the acronyms used for the major categories of output values. Although not shown, each output value is further categorized based on part of speech (MIP@Adj, SPS@Adv, WNS@Noun, and so on):

Acronym	Meaning
MIP	MinorProblem
WPS	WeakPositiveSentiment
SPS	StrongPositiveSentiment
WNS	WeakNegativeSentiment
SNS	StrongNegativeSentiment
NES	NeutralSentiment
COR	ContactRequest
GER	GeneralRequest

In addition to the above acronyms for the thesaurus entity categories, the following are used by some languages:

Acronym	Meaning
WPP	for keywords extracted as weak positive sentiment unless negated in which case they are extracted as minor problem (and not weak negative sentiment), such as “easy”, ergonomic”, “robust”
MIPn	for keywords extracted as minor problem only when negated, such as “hygienic”, “utility”. No extraction otherwise.
WNSn	for keywords extracted as weak negative sentiment only when negated, such as “purpose”, “sense”, “value”. No extraction otherwise.
SNSc	for keywords extracted as strong negative sentiment when in context (i.e., together) with other unequivocal sentiment keywords, such as “amarissimo”, “durissimo”. No extraction otherwise. In Italian only.
SPSc	for keywords extracted as strong positive sentiment when in context (i.e., together) with other unequivocal sentiment keywords, such as “dolcissimo”,”fortissimo”. No extraction otherwise. In Italian only.
WNSc	for keywords extracted as weak negative sentiment when in context (i.e., together) with other unequivocal sentiment keywords, such as “amaro”,”duro”. No extraction otherwise. In Italian only.

Acronym	Meaning
WPSc	for keywords extracted as weak positive sentiment when in context (i.e., together) with other unequivocal sentiment keywords, such as “dolce”,”forte”. No extraction otherwise. In Italian only.
TKN	for multiword tokens that are not keywords, such as “a buon fine”, “a mio parere”. It is typically used for fixed expressions: declaring them as TKN reduces the possibility of them being wrongly analyzed as topics.

Modifying and Activating the Dictionary

Entries can be added, removed or changed under each entity category.

- To add new keywords: add the entry under the desired output value and part of speech.
- To remove keywords: either delete or comment out the keyword you don’t want.
- To change the output value of keywords: use cut-and-paste to move keywords to the desired entity category.

When you have finished editing the sentiment keyword dictionary, you must save it and compile it..

i Note

It is very important that new or modified keywords are assigned the correct part of speech. Entries from the dictionary get compiled and picked up by the Sentiment Analysis rules. If a new keyword is not added to the correct part of speech category, you may not get the desired behavior.

It is also important not to change the names of the entity categories as the rules expect these names in order to work properly.

We recommend making a copy of the delivered sentiment keyword dictionary and editing it, never altering the original.

3.5.2 Nonwhitespace Language Sentiment Analysis Customization

Because of the nonwhitespace nature of Chinese, customization is done slightly differently for simplified and traditional Chinese.

Format of Customizable Dictionary

Five categories can be customized is simplified and traditional Chinese:

- CustomTopic
- CustomPositive
- CustomNegative

- CustomNeutral
- CustomProblem

The following shows the contents of the simplified Chinese dictionary as an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dictionary>
    <entity_category name="CustomTopic">
        <entity_name standard_form="物流">
            <variant name="快递"/>
            <variant name="邮递"/>
        </entity_name>
        <entity_name standard_form="洗发产品">
            <variant name="洗发水"/>
            <variant name="洗发露"/>
            <variant name="洗发膏"/>
            <variant name="潘婷洗发膏"/>
            <variant name="海飞丝"/>
        </entity_name>
    </entity_category>
    <entity_category name="CustomPositive">
        <entity_name standard_form="一直用">
            <variant name="一直使用"/>
            <variant name="一直在用"/>
        </entity_name>
        <entity_name standard_form="高档">
            <variant name="高端"/>
        </entity_name>
    </entity_category>
    <entity_category name="CustomNegative">
        <entity_name standard_form="低档">
            <variant name="低级"/>
        </entity_name>
    </entity_category>
    <entity_category name="CustomNeutral">
        <entity_name standard_form="中档">
        </entity_name>
    </entity_category>
    <entity_category name="CustomProblem">
        <entity_name standard_form="干燥">
            <variant name="干裂"/>
            <variant name="干枯"/>
        </entity_name>
    </entity_category>
</dictionary>
```

Modifying and Activating the Dictionary

Entries can be added under each entity category, with their standard forms and variant forms. Entity names and/or their various forms in the sample dictionary can either be commented out or replaced, as necessary.

When you have finished editing the sentiment keyword dictionary, you must save it and compile it..

Note

It is important to follow the syntax when adding entries.

It is also important not to change the names of the entity categories as the rules expect these names in order to work properly.

We recommend making a copy of the delivered sentiment keyword dictionary and editing it, never altering the original.

4 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**.

4.1 About Customizing Extraction

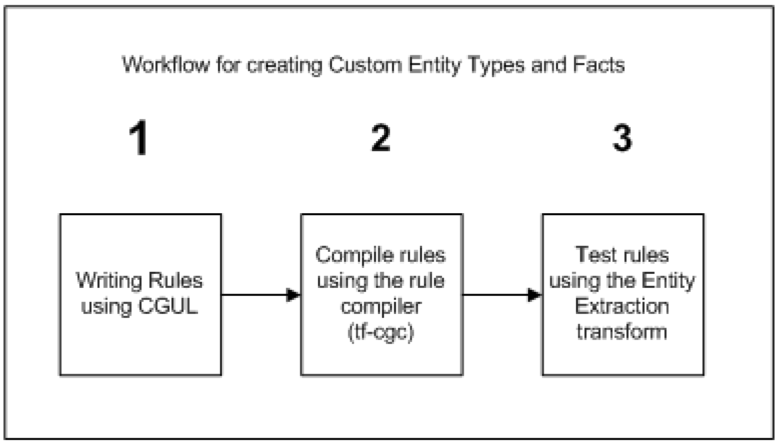
The software provides tools you can use to customize extraction by defining extraction rules that are specific to your needs.

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.

The following diagram describes a basic workflow for testing extraction rules:


Workflow for creating Custom Entity Types and Facts
1
2
3
Writing Rules using CGUL
Compile rules using the rule compiler (tf-cgc)
Test rules using the Entity Extraction transform

Related Information

Preparing Extraction Rules for Use [page 68]

4.2 Understanding Extraction Rule Patterns

Using 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*.

4.2.1 CGUL Elements

Provides an overview of the elements of CGUL rules.

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 [page 35].	

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 [page 44].	

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 [page 59].
	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 [page 53].

Element	Description	
	Grouping and Containment Operators	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 [page 62].
	Expression Markers	These include the following markers: `[SN]` – Sentence `[NP]` –Noun phrase `[CL]` and `[CC]` –Clause and clause container `[OD]` –Context `[TE]` –Entity `[P]` –Paragraph For more information, see Expression Markers Supported in CGUL [page 47].
	`Match` Filters	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 [page 65].

Element	Description
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 [page 32].
Lexicon Directive	#lexicon directives are used to include the contents of a lexicon file that contains a list of single words, delimited by new lines. For more information, see Including a Lexicon in a Rule File [page 33].
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.

4.2.2 CGUL Conventions

Spells out the syntax conventions for CGU rules.

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 Information

Character Classifier (\p) [page 57]

4.3 Including Files in a Rule File

Discusses the syntax for rule sets.

You can reference CGUL source files and .pdc files within your rules file. All #include and #lexicon directives should precede all other types of lines in the file. The keyword symbol -- # -- must be the first nonspace character on the line.

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

i 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. .

You can use an absolute or relative path for the file name, but it is recommended that you use an absolute path.

i 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.

4.3.1 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.

4.4 Including a Lexicon in a Rule File

You can include a lexicon within your rules file. All `#lexicon` directives should be placed along with all `#include` directives at the top of the file.

The keyword symbol -- `#` -- must be the first nonspace character on the line

The lexicon file consists of single words separated by new lines. The compiler interprets the contents of a lexicon 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 lexicon.

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

Example

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

Description

In this example, the lexicon is compiled as a `#define` directive named `FRUITLIST` and is contained in a file called `myfruits.txt` (see Including a Lexicon in a Rule File in the *Data Services Designer Guide*). Later in the rule file the lexicon 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>
```

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

If multi-word entries need to be added, a combination of `#lexicon` and `#subgroup` directives can be used:

```
#lexicon FRUITLIST "myfruits.txt"
#subgroup FRUITLIST2: (<wild> <STEM:cherry|strawberry>)|(<white> <STEM:peach>)
#group FRUIT: <STEM:%(FRUITLIST)>|%(FRUITLIST2)
```

4.5 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 Information

CGUL Conventions [page 32]
Writing Directives [page 35]
Using the #define Directive [page 36]
Using the #subgroup Directive [page 37]
Using the #group Directive [page 38]
Using Items in a Group or Subgroup [page 43]

4.5.1 Writing Directives

Directives can be contained in one input line or can span more than one.

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 `{}`.

4.5.2 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`

i 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.

If multi-word entries are needed, #subgroup or #group directives are more useful:

```
#define COLOR: (red|blue|white)
#subgroup COLORS: <navy|sky|light|dark><blue>
#group MYCOLORS: <%(COLOR)>|%(COLORS)
```

Related Information

Using Predefined Character Classes [page 33]

4.5.3 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>`

i 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**, **Rochert**). 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

4.5.4 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>.`

i Note

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

i Note

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

There is no "longer match win" rule for #group directive. If the same rule file contains the following #group directives:

```
#group A: <A>
#group AB: %(A) <B>
#group ABC: %(AB) <C>
```

and the input is "A B C", all 3 will match resulting in the following output:

```
- A
- A B
- A B C
```

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>
#group <DROP_name> : <expression>
#group <AA_name>: <expression>
#group <ZZ_name>: <expression>
```

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

Parameter	Description	Example

`name`	The name you assign to the extracted fact or entity	`#group Positive: <expression>`
		
		"Positive" is the name of the extracted fact.
		
		Two types of prefixes can be added to #group names each which different results:
		
		• `DROP_`: Use the "DROP_" prefix for expressions you do not want to extract.
		
		`#group DROP_Positive: <If> <expression>`
		
		In this example, any sentence starting with if will be “dropped” and removed from the results.
		
		• `AA_` or `ZZ_`: These prefixes are used to prioritize matches of same length. "AA_" has the highest priority, "ZZ_" the lowest. No prefix is the normal priority.
		
		`#group AA_Positive: <STEM:love> <>* <POS:Nn>`

Parameter	Description	Example

`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> | %(BallSports)`
		
		or
		
		`#group BodyParts: <head> | <foot> | <eye>`
		
		`#group Symptoms: %(BodyParts) (<ache>|<pain>)`
		
		In this example, the `BallSports` 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`.

`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.

Parameter	Description	Example

`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

`<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.

4.5.5 Using Prefixes in a #group Directive

Two types of prefixes can be added to the names of #group: `DROP` and Priority indicators: `AA_` or `ZZ_`

DROP Prefix

The rule writer can specify a version of an entity that should be dropped if it wins overlap resolution. This feature has been provided because it can sometimes be simpler and more efficient to overextract and then discard certain entities. The idea is to first overextract an entity by using a rule that is too general, but then to also provide an additional rule to discard an undesired version of the entity in a final pass.

Usage: To indicate a droppable entity, give its name a `DROP_` prefix, as in `DROP_Sentiment`.

Behavior:

- CGUL entity overlap resolution will include `DROP_name` when comparing `name` entities.
- If the extraction span (offset and length) is not the same, then normal-style overlap resolution is done as if the prefix was not present.
- Otherwise (same span), droppable entities win over non-droppable entities (assuming no priority difference - see next section).
- After overlap resolution, if the winning entity has the `DROP_name` prefix, then the entity is discarded.

Example

```
! DROP_ feature allows droppable version of entity to win overlap resolution
and then itself be dropped.
! Input: "I don't want a catalog"
! 2 matches: MyRequest = "want a catalog", DROP_MyRequest = "n't want a
catalog"
! 0 result in display:  neither of the matches, since the longest
DROP_MyRequest wins and is then dropped
#group MyRequest: [OD GeneralRequest]<STEM:want>[/OD] [OD Topic][NP]<>*[/NP][/
OD]
#group DROP_MyRequest: <STEM:not> <>*
```

AA and ZZ Prefix

The rule writer can also specify a simple priority system for different versions of an entity. Like droppable entities above, this feature can lead to simpler and more efficient rules. This priority is only used to resolve ties, i.e. extractions with the same span (offset and length).

Note

If both `DROP_name` and priority indicators (`AA_` and `ZZ_`) are used, `DROP_name` must come before priority, for exxample. `DROP_AA_Sentiment`. In this case, priority has precedence over `DROP_name`.

Usage: Use an `AA_` prefix to indicate higher than normal priority or `ZZ_` for lower than normal priority.

Behavior:

- CGUL entity overlap resolution will include `AA_name` and `ZZ_name` when comparing name entities.
- If the extraction span (offset and length) is not the same, then normal-style overlap resolution is done as if the prefix was not present.
- Otherwise (same span), if the priority is different, then `AA_name` wins over name (or `ZZ_name`) and name wins over `ZZ_name`.
- Otherwise (same span and priority), droppable entities win over non-droppable entities.
- The priority prefixes are stripped from the final results.

Example

```
! AA_/ZZ_ feature specifies priority for entity overlap resolution when
location and length are same.
! Input: "not only nice"
! 2 matches: Test (from Test) = "not only nice" and Test (from AA_Test) =
"not only nice"
! 1 result in display:  Test (from AA_Test) = "not only nice"
#group Test: <not> <POS:Adv> <nice>
#group AA_Test(priority="AA"): <not> <only> <nice>
```

4.5.6 Using Items in a Group or Subgroup

You can use an item defined by any <ph translate="no" xml:lang="en-US">CGUL</ph> 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` need not 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)>
```

4.6 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.

4.6.1 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.

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.

  > **i 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. For the complete list of available part-of-speech tags in each language, please refer to the *Language Reference Guide*.
- 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**.

Consult your *Language Reference Guide*, the Stemming and Part-of-Speech tagging topics.

4.7 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 `[OD]` (Output Delimiter) and `[TE]` (Entity Marker), can use key-value pairs to specify attributes. 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] expr [/NP])`	Specifies the exact range of an expression `expr` that is a noun 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.
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.

Operator	Description
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.

Related Information

Paragraph Marker [P] [page 48]
Sentence Marker [SN] [page 48]
Noun Phrase Marker [NP] [page 49]
Clause Marker [CL] [page 49]
Clause Container [CC] [page 49]
Context Marker [OD] [page 50]
Entity Marker [TE] [page 51]
Unordered List Marker [UL] [page 52]

4.7.1 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`, or `TE` markers

4.7.2 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`, or `TE` markers
- It is not used in pairs

4.7.3 Noun Phrase Marker [NP]

Use the noun phrase marker `[NP] 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]
```

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` or `TE` markers
- It is not used in pairs

4.7.4 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, Arabic, Dutch, Farsi, French, German, Italian, Korean, Simplified Chinese and Spanish.

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`, or `TE` markers
- It is not used in pairs

4.7.5 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, Arabic, Dutch, Farsi, French, German, Italian, Korean, Simplified Chinese and Spanish.

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`, or `TE` markers
- It is not used in pairs

4.7.6 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 as well as `exp` (between the `[OD]` markers). Both spans are displayed in the output, with the `OD exp` getting its own label.

In the following example, any proper nouns preceded by either `Sir` , `Mr` , or `Mister` are matched. The resulting patterns shown in the output are both the whole span and the OD span, that is, "Mister Brown/Name" and "Brown".

```
#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. The resulting patterns shown in the output are both the whole span and the OD span with label "LastName", that is, "Mister Brown/Name" and "Brown/LastName".

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

i 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 Information

Writing Extraction Rules Using Context Markers [page 52]

4.7.7 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 or a custom-defined entity from a dictionary 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

Entities defined in user dictionaries can be referenced through `[TE]` markers.

For example, a user-defined dictionary contains the entity category TOYS:

```
<?xml version="1.0" encoding="windows-1252" ?>
<dictionary>
<entity_category name="TOYS">
    <entity_name standard_form="Lego" />
    <entity_name standard_form="PlayMobil" />
</entity_category>
</dictionary>
#group Play: <STEM:play> <>* [TE TOYS] <>+ [/TE]
```

4.7.8 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.

i 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

4.8 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.

4.9 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

4.9.1 Standard Operators Valid in CGUL

The standard operators are the wildcard, alternation. escape, character negation, string negation, subtraction, character classifier.

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

Operator	Symbol	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.
Escape	\	Escapes special characters so they are treated as literal values.

Operator	Symbol	Description
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 Information

Character Wildcard (.) [page 54]
Alternation (|) [page 55]
Escape (\) [page 55]
Character Negation(^) [page 56]
String Negation(~) [page 56]
Subtraction (-) [page 57]
Character Classifier (\p) [page 57]

4.9.1.1 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.

i Note

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

4.9.1.2 Alternation (|)

The alternation operator (`|`) acts as a Boolean OR in regular expressions, allowing 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.

> **i 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.

4.9.1.3 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>`.

> **i Note**
>
> Only valid when preceding a special character.

4.9.1.4 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
```

> **i 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.

> **i Note**
>
> Character classifiers do not operate with character negation.

4.9.1.5 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)
```

> **i 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.

i 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.

i Note

Character classifiers do not operate with string negation.

4.9.1.6 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
```

In the following French example, subtraction is used in `#define` in order to subtract adverbial forms (incroyablement) from the adjectival forms (incroyable):

```
#define roots: (incroyabl|superb)
#define pos_adv: %(roots)(ement)
#define pos_adj: (%(roots).+) - %(pos_adv)
#group PosSentiment: <POS:Nn> (<%(pos_adj)>|(<%(pos_adv)> <POS:Adj|V-PaPart|V/
Adj>))
```

The above pattern will extract "un dessin incroyable" and "un dessin incroyablement coloré".

i Note

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

4.9.1.7 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)

i 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, any of the following expressions will 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 **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

4.9.2 Iteration Operators Supported in CGUL

The iteration operators are question mark asterisk, plus, and braces.

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

Operator	Symbol	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

i 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 Information

Question Mark (?) [page 60]
Asterisk (*) [page 60]
Plus Sign (+) [page 60]
Braces ({ }) [page 61]

4.9.2.1 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**, **Gaddaafiy**, **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

4.9.2.2 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

4.9.2.3 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

4.9.2.4 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

i Note

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

i Note

If exact iteration larger than 9 is needed, this can be expressed as a range starting and ending with the same number:

```
{10,10}
```

means 10 iterations.

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 of 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**.

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

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

Related Information

Range Delimiter (-) [page 63]
Range Operator ([]) [page 63]
Item Grouper ( ) [page 64]

4.9.3.1 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.

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

4.9.3.2 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.

> i Note
>
> Blank spaces within the range operator will cause a syntax error.

> i Note
>
> 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]
```

i 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**.

4.9.3.3 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)xide> <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.**

4.9.4 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)

Functions	Operators
Alternation	|

4.9.5 Special Characters

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

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

4.10 Match Filters Supported in CGUL

CGUL supports the "longest match", "shortest match", and "list" (all matches) filters.

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 Information

Longest Match Filter [page 66]
Shortest Match Filter (?) [page 66]

List Filter (*) [page 67]

4.10.1 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` and `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.

i 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 Information

Match Filters Supported in CGUL [page 65]
Shortest Match Filter (?) [page 66]
List Filter (*) [page 67]

4.10.2 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".

i Note

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

Related Information

Match Filters Supported in CGUL [page 65]
Longest Match Filter [page 66]
List Filter (*) [page 67]

4.10.3 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

i 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 Information

Match Filters Supported in CGUL [page 65]

Longest Match Filter [page 66]
Shortest Match Filter (?) [page 66]

4.11 Preparing Extraction Rules for Use

When you have 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.

i 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 (byte order 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`.

5 CGUL Best Practices and Examples

5.1 Best Practices for Rule Development

These are the guidelines to follow 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 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 3
   #define chemical_prefix:
   {
   ((mono)|
   (bi)|
   (di)|
   (tri)|
   (tetra)|
   (octo)|
   (deca)|
   (hepto)|
   (desoxy)|
   (mero)|
   (meso)|
   (ribo))
   }
   !example 4
   #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` statement 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) [page 57] 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, especially 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. Avoid writing: `#group quote_sent: <>* [TE PERSON]<>+[/TE] %(say_expr) <>*`
    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. For more information, refer to the*Text Data Processing Language Reference Guide*.
16. 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)>`.

17. 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.

5.2 Common Syntax Errors in Custom Rules

Certain common syntax errors 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`.

5.3 Examples For Writing Extraction Rules

To write extraction rules (also referred to as CGUL rules), you must first determine 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 Information

Example: Writing a simple CGUL rule: Hello World [page 74]
Example: Extracting Names Starting with Z [page 74]
Example: Extracting Names of Persons and Awards they Won [page 75]

5.3.1 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.

5.3.2 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. 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.

5.3.3 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: <Best|Top> <POS:Num>? <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 expression that includes combinations of either **Best** or **Top** followed by an optional number and one or more proper nouns. For example, **Top Ten Entertainers**, **Best Footballer Players**, 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**.

6 Testing Dictionaries and Extraction Rules

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

Important Disclaimers and Legal Information

www.sap.com/contactsap

THE BEST RUN SAP