{gender*render}

Specification

Template system and implementation specification for rendering gender-neutral email templates with pronoun information

phseiff
from phseiff.com
v0.6.0

March 24, 2021

Contents

1 Abstract

Our society, as well as the way we perceive gender, are steadily evolving. This evolution does not hold in light of technological questions, and it is our -the "IT people"'s duty- to address and do our best to solve the social issues that arise from our technology. One such technology are email- and other text templates, which are becoming increasingly popular to automate customer interactions of any kind, be it in newsletters, notifications or program menus. Many such templates are gender-specific, in that they address the reader in a gendered fashion ("Dear Mrs. Dursley, ..."). Such templates are relatively easily implemented by providing two versions of the email, one for every binary gender. However, some texts are far more complicated, because they address multiple people (each with their own gender unknown at the time of writing), or people in the third person (throwing their pronouns into the mix). In addition, an increasingly height amount of people uses non-binary pronouns, or gender-neutral pronouns, many of whom might not yet be discovered at the time of writing, which makes these people marginalized when it comes to being correctly addressed even in automated emails.

Similar issues exist in electronic entertainment industries, namely the computer game industry. Many computer games, especially RPGs, allow players to choose their gender, thereby altering the dialogues they receive from the game in regards to pronouns and other grammatical implications of gender. Since being inclusive towards different identities moves more and more into focus in the industry, the need arises to allow players to choose from more than just two genders, which, in its most inclusive form, requires players to be able to customize their pronoun preferences in the character creation menu.

This creates the requirement for creating template systems for the english language, and, in extension, any natural language (since all languages work differently), that support writing complex texts in a gender-neutral fashion and later "render" them to correctly gendered texts, given the pronoun preferences of every person they refer to.

gender*render is an attempt at creating one such template language, including a Specification, to serve as a proof of concept as well as a starting point for people who want to implement similar things. The vision behind this proof of concept is not only to show *how* addressing people with unconventional preferred pronouns can be automized, but also to show *that* it can be easily automized, to debunk the myth that properly addressing non-binary people in an automated fashion is simply technically impossible.

In essence, gender*render accepts a gender-neutral template for a text in a specific syntax with special annotations on how to render it into a gender-neutral version, as well as data describing the pronoun preferences of all people

it refers to, and returns a correctly gendered text from it; a task less trivial than one might assume at first glance.

2 Requirements

There are multiple requirements for such a template language, whom I will list here, including short explanation of why they are required wherever I deem it necessary:

- The language must be easy to use even for less tech affine people. This means that the atoms of the language, such as tags et cetera, must be as short as possible, and should not clash with commonly used words or signs, so the amount of escape characters the user needs to use is minimal.
- The language must support different scenarios:
  - One person being addressed versus multiple people being addressed
  - Only people mentioned in first person, only people mentioned in third person, or a mixture of both
  - Everyone using pronouns versus some people preferring not to use any pronouns
- The fact that multiple scenarios are supported may not make using the template language for only a subset of them more complicated that it needs to be.
- Rendering templates may only require the information needed for rendering the template. For example, rendering a template that never addresses anyone in the first person should not require providing information as to whether the person goes by "Mr", "Mrs" or any other form of address. This is especially relevant since users do not want and should not need to provide more information that necessary for rendering the templates, especially considering the intimate nature of preferred pronouns.
- The syntax should be describable using a context-free grammar in conjunctive normal form, which allows easy syntax checking and syntax highlighting.
- The data containing a persons preferred pronouns should be given in a widely-used, standardized format, such as JSON.

3 Design Decisions

The following decisions where made based on the the technical requirements ruled out in the corresponding section:

- The language uses a syntax similar to pythons build-in string formatting syntax, using curly brackets to annotate gender-specific parts of a sentence. Backslashes are used as escape characters for the rare occurrences where curly brackets are actually needed.
- In addition to grammatical terms (e.g. "possessive pronoun"), using the gender-neutral form (e.g. "their") in tags is supported, potentially making texts more fluid to write and easier to read in their un-rendered form.
- If tags contain IDs to annotate which person is referred to, a mapping of IDs to pronoun preferences is accepted for rendering. If no such IDs are added to the document because only one person of unknown gender is addressed in the document, the pronoun preferences are directly accepted by the renderer, without having to be part of a person-to-pronoun-mapping. This supports referring to multiple persons in one text without making the writing of texts that refer to only one person any more troublesome.
- The pronoun information is given to the renderer as a piece of JSON data (or a similar object if the language used by the implementation supports such objects, e.g. dicts in Python, but strings of JSON data should always be supported). Information that is not required by the template may be left out in the template.
- Templates can be parsed before being rendered and then used for multiple renderings. This should debunk the idea that gender-sensitive template systems are to inefficient to use them.

These design decisions contain only those that are relevant to the requirements listet in the previous section; in-depth explanation and definition of the way the template system works are given in the next section.

4 Standard

This section contains the actual standard. It is divided into three subsections; one for defining the template language and how gender-neutral texts are described with it, one for defining the data structure used to describe the pronoun preferences of all people mentioned in a template, and one for guidelines and specification on implementing a renderer for the template language.

The terms "MUST", "MUST NOT", "SHOULD", "SHOULD NOT" and "MAY" in this document are used as defined by the RFC 4627. Additionally, the term *does* implies that it *must do*, not that it *can do*.

4.1 Template Language

Any text that follows the syntax of the following definition is considered a valid gender*render -template. Any text that does not follow the following

is not considered a valid gender*render -template. Files whose content is a valid gender*render -template are referred to as files containing gender*render -templates in the following section, and *not* as gender*render -templates on their own. It is recommended to save such files with the file type `.grt` (short for "gender render template").

The purpose of gender*render -templates is to write texts in a gender-neutral way (at least in regards to some of the individuals they refer to), and to be valid input for the gender*render -renderer, which is described in a later section.

gender*render -templates may contain an arbitrary number (including zero) of gender*render -tags. A gender*render -tag is defined a sequence of characters that starts with an unescaped left curly bracket ("{", `U+007B`) and ends with an unescaped right curly bracket ("}", `U+007D`) without containing any unescaped curly brackets (`U+007B` as well as `U+007D`) in between. The purpose of gender*render -tags is to describe gender-specific sentence components in a gender-neutral fashion, these usually being mentions of a person in the third person singular.

A character is considered escaped if it is proceeded by an unescaped backslash ("\", `U+005C`) or by a backslash which is not proceeded by other backslash. A backslash which is not escaped is called an escape-character. A template which contains backslashes which are neither escaped nor escape characters is not considered a valid gender*render -template, as is any template which contains unescaped curly brackets who are not part of any valid gender*render -tag. A template which ends with an escape character, which isn't followed by a character to escape, is also considered invalid.

Every character of a gender*render -tag except the first and last characters (the brackets) is considered part of its content. Said content is divided into sections through unescaped asterisks ("`*`", `U+002A`). A section of a gender*render -tag does not contain any unescaped asterisks, and it must contain at least one non-whitespace[1] character. Colons ("`:`", `U+003A`) are considered special characters in sections, and may thus appear at most once per section, and neither as the first nor as the last non-whitespace character of the section. If a section contains a colon, the characters of the section beforehand the colon (minus all leading or trailing whitespace) are called the sections *type descriptor*, and the characters on the right side of the colon (minus all leading or trailing whitespace) are called the sections *values*, where every whitespace-separated word is one value, with the order mattering. If a section does not contain a colon, all of its content is considered part of its value-section. Whether a section type allows multiple values or just one depends on the section type.

[1] "Whitespace" as defined by the HTML Living Standard.

Colons proceeded by an escape character are not considered to be special chars, and instead behave like any other character; analogously, escaped whitespace behaves like any other character and thereby indirectly enables the usage of whitespace in section values as well as section types. Individual section values as well as section types may therefore not contain whitespace, but may very well contain escaped whitespace, which is then parsed into "normal" whitespace by the parser.

There are multiple different types of section, assigned to sections by their type descriptor. A section whose type is "foo" is called a "foo-section". Every type of section has a unique priority, as a real number between 0 and 1000, assigned by this specification. After parsing a tag, its right-most section with no type descriptor and no assigned section type is automatically assigned the section type with the highest priority of all section types that are not assigned to any section of the tag (by this rule or a section type descriptor) yet; this calculation is then repeated for every type-less section of the tag from right to left. Every gender*render -tag must have at least one section, and may only have one section of every type; this takes into account the assigned section type of sections without a type descriptor. In addition, a tag may not contain more sections than there are section types defined by the spec.

The most basic type of section is the `context`-type, which describes the syntactic context (what grammatical case or attribute of a person it stands for) of the gender*render -tag. Every gender*render -tag must have one context-section.

There is an infinite amount of possible context values (values that a tag's context section may have), only a subset of whom is hard-coded into this specification (the other ones are evaluated and interpreted at runtime). These values are referred to as *specified context values*. Some specified context values are referred to as *canonical context values*, whilst the others are aliases of canonical context values, which are interpreted identical to the canonical context value they correspond to.
Whilst the canonical context value is usually the most descriptive of its aliases, it is not necessarily recommended to actually use it in a gender*render -template. Instead, every canonical context value has an alias referred to as its *recommended alias*, which is the least descriptive of its aliases, but fits into the flow of every text, as it imitates an exemplary gender-neutral version of the thing the context value represents; e.g. rather than saying "{subject} did all the hard work {reflexive}", one can say "{they} did all the hard work {themself}" by using the recommended aliases.
Some non-specified context values are regarded as canonical context values as well, but (contrary to how specified context values behave), not every non-specified context value that is also non-canonical is automatically an alias to a canonical non-specified context values, since some are simply evaluated outside the scheme of canonical context values and their aliases.

When applied to a tag (e.g. "the tag's canonical context value"), the canonical context value refers to the canonical context value of which the context value of the tag is an alias, or the context value of the tag itself, in case it is a canonical context values.

Another important distinction to be made in regards to different types of context values (this one important when it comes to actually rendering the template) is between *direct-mapped* and *non-direct-mapped* context values. Tags with direct-mapped context values are (usually) simply replaced with the corresponding attribute of the pronoun data of the person the tag corresponds to [2] during rendering. For example, the tag "context:foo" will be replaced with the "foo"-attribute of the pronoun data of the person that the tag corresponds to, as will "context:bar" if "bar" is an alias for "foo", assuming "foo" is a direct-mapped context value. What tags with context values that are not direct-mapped are rendered to, on the other hand, depends on logic that is executed on rendering time based on the context value of the tag and pronoun data of the individual the tag corresponds to. Note that there are some exceptions to this rules; some context values that are referred to as direct mapped context values still perform some additional tests before being rendered in the expected direct-mapped way or behave different in some cases, such as the "*address*"-context value. The distinction between direct-mapped and non-direct-mappe context value is ultimately made by the specification on an individual and conceptual basis rather than strictly according to this rule of thumb.
Every specified context value from this main specification is direct-mapped (though the oppositete does not apply), but please note that extension specifications [3] may specify specified context values that are not directly mapped. It's also worth noting that direct-mapped context value always fall into the scheme of canonical context values and their aliases, though the opposite does not necessarily apply to all extension specs.

Please note that the distinction between canonical context value, recommended alias and other aliases is hardly relevant to the actual user, since all they need to know is that there are multiple possible values for each grammatical context; the reason why the concept of canonical context values (as opposed to simply listing multiple coequal values for every context) is included in the specification is that it is (a) useful for implementations, and (b) to conceptualize this central design pattern of the gender*render template language, which can be found everywhere throughout the specification. This can be analogously applied to the concept of specified/ unspecified and direct-mapped/ not direct-mapped context values.

The following table lists the possible values a `context`-section's value may

[2] How tags correspond to individuals depends on their "id"-value, which is explained in-depth below.

[3] refer to the *Specification Development* section to learn about extension specifications.

have, as well as their meanings, though the syntactic validity of the template does not depend on whether the values and types of the the gender*render -tags are listed in this specification. For specified context values, the first value listed for every context (bold) is the recommended one, whilst the last one (italic) is the canonical one:

syntactic context indicated by the value(s)	possible values, synonymous to each other	short explanation, where necessary
Specified context values that refer to grammatical cases of pronouns:		
Subject	**they**, subj, *subject*	
Object	**them**, obj, *object*	
Dependant possessive Determiner	**their**, dposs, *dpossessive*	
Independent possessive Determiner	**theirs**, iposs, *ipossessive*	
Reflexive	**themself**, reflex, *reflexive*	
Specified context values that refer to non-grammatical attributes of a person:		
Form of Address	**Mr_s**, Mr, Mrs, *address*	
Surname	**Doe**, name, family-name, *surname*	(This is a reference to "John Doe", a commonly used placeholder for names.)
Personal name	**Joan**, first-name, *personal-name*	(This is, analogously to "Doe" for family names, a reference to "John Doe", but "Jean" is gender-neutral and thus makes for a much better placeholder than "John".)
Non-specified, yet direct-mapped context values:		
Custom property	"<" *property* ">" (Note that "<" *property* ">" is considered its own canonical context value, even though it has no aliases.)	*property* can be any string without whitespace, and refers to a property of an individual that is defined by its pronoun data as a string, yet not part of the spec.
Non-specified non-direct-mapped context values:		

Gender-specific Noun	any nominative, with whitespaces replaced by underscores ("_", `U+005F`) (note that custom properties take precedence over gender-specific words, so context values only qualify as this type of context value if they don't qualify as a custom property.)	If the value of the `section` does not match any of the above, its content is understood as being a noun which either server as a substitution or as a description of a person. For example, the sentence "{name} is an {actor}" or "the {actor} asked for applause" would be good candidates for using said type of value since "actor" has two different gendered forms ("actor" and "actress") in english.

Table 1: Types of context values in tags

The priority of the `context`-section type is `1000`. If the `context`-section of a tag contains multiple values (e.g. "{`foo:bar * context:Mr_s Doe`}"), the tag is interpreted as a sequence of multiple different tags separated by single spaces (" ", `U+0020`) that differ only in their context value and are identical to the original tag in every other aspect (e.g. "{`foo:bar * context:Mr_s`} {`foo:bar * context:Doe`}").

Context values are always interpreted as being lower-case (all characters with lower-case versions, at least), meaning no context value with upper-case characters will ever be specified by the specification, and the rendering process interprets every context value as if it was entirely lower-case (capitalized values on the table above are capitalized merely for readability). This includes custom properties ("<*property*>") in gender*render -templates, which are interpreted as their lower-case equivalent as well, in case they contain upper-case letters. Should a context value in a template be capitalized, this capitalization is interpreted as syntactic sugar to describe the tag's capitalization-value (a type of context section defined in the next paragraph) and the context value will be converted to lower-case in the parsing process.

The second (and priority-lowest) section type supported by this version of this Specification is the `capitalization`-type, whose priority is `800`. The value of the `capitalization`-section describes how the word(s) to which the tag will be resolved will be capitalized, and may only take a limited set of values, each of whom corresponds to a different type of capitalization. The potential `capitalization`-values of a tag are listed in the following table:

value	example (foobar)	short explanation, where necessary
lower-case	`foobar`	
capitalized	`Foobar`	
all-caps	`FOOBAR`	
studly-caps	`FoObAr`	every second letter lower-case, all other upper-case.
alt-studly-caps	`fOoBaR`	like studly-caps, but reversed.

Table 2: Possible capitalization-values and their meaning

The example-column of the table above illustrates the capitalization that will be applied to tags with the corresponding `capitalization`-value after they are resolved. If a tag's capitalization value is specified explicitly, whilst its context value is not entirely lower-case, the gender*render -template is invalid; if, on the other hand, the capitalization value is omitted, and the context value is capitalized according to one of the examples of the table above (which is the recommended way to specify a tag's capitalization value), then the capitalization value will be set to the corresponding value and the context value will be converted to lower-case during parsing[4]. A tag "{`FoObAz`}", for example, will become "{`capitalization:studly-caps*foobaz`}" during parsing. This modification is applied *after* the aforementioned splitting of tags with multiple context values into series of tags, to allow a more fine-grained customization of tag capitalization; therefore, "{`Foo bar`}", for example, will become "{`capitalization:capitalized*foo`} {`capitalization:lower-case*bar`}".
When determining whether a string follows a specific type of capitalization, characters that are neither lowercase nor uppercase are interpreted as capitalized in accordance with the capitalization type that is being checked for (an important distinction because another way to implement this would be to skip the character and proceed with the next character as if there had not been any character in between); whether a character is uppercase, lowercase or neither is decided in accordance to the Unicode Standard.
If a tag's context value is capitalized in a different way than the ones listed in the example-column of the table above, the gender*render -template is invalid; please note, however, that this behavior might change in the future in any minor release in case a fallback-value gets introduced and is therefore technically undefined, which implementation documentations should properly communicate to the user[5]. Templates with tags with a capitalization value which differs from all off the above-defined are invalid as well.
In cases where the capitalization of a tag's context value is ambiguous (like "Fo", which could be interpreted as both `capitalized` as well as `studly-caps`), the topmost of both possible interpretations, according to the order of the table

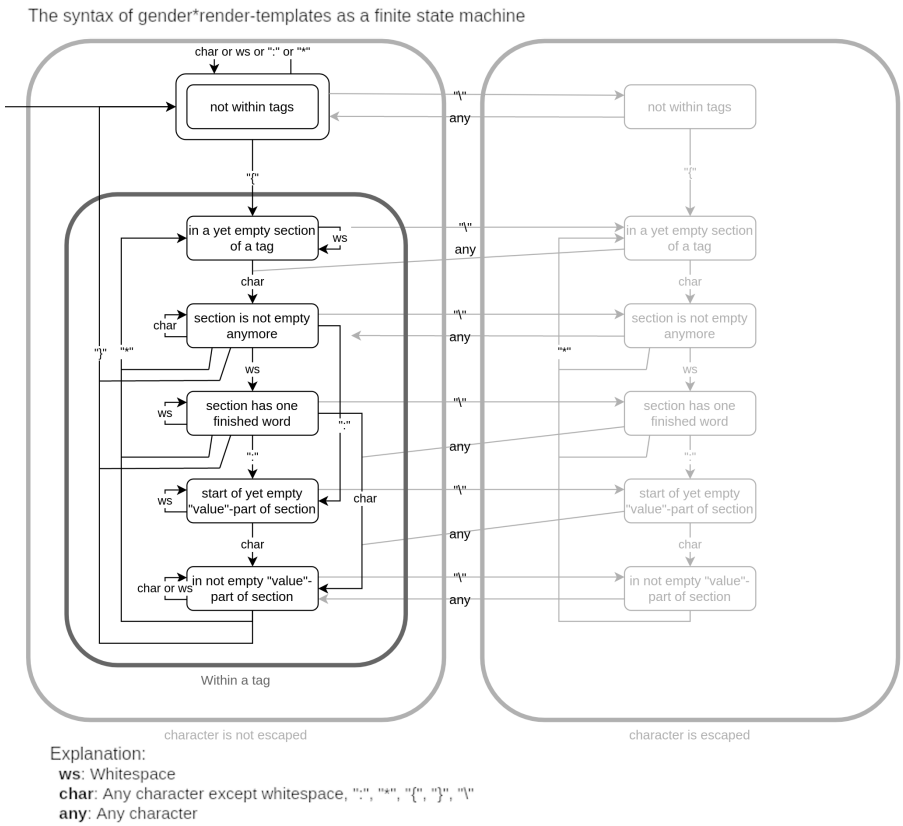
[4]This uses the lower-casing algorithm described in section 3.13 of the Unicode Standard.
[5]See this comment on GitHub for discussion on this.

above, takes precedence.

The third section type supported by this version of this Specification is the `id`-type, whose priority is `950`. `id`-sections may take any value, as long as they take only one value, including (but not recommended to users for obvious reasons) values that contain escaped whitespace. Said value describes which individual the gender*render -tag refers to. Two gender*render -tags with the same id-value therefore refer to the same individual. The id-value can be omitted by the user if there is only one individual mentioned in the whole template, and in some other cases; this is explored further in the renderer section. Whether there is an `id`-section is not part of the template specification, since it is not clear until the pronoun information is given.

Since there are only three section types defined by this specification, the priority-highest of them being mandatory and the priority-lowest preferably being described using semantic sugar, there is no practical need to use any section descriptors. They are still defined as a language feature in this template to provide a way to port the template language to other natural languages that might require additional information without having to introduce new syntax elements for every language. Please keep in mind that further extension specifications will break backwards compatibility if they introduce section types with priority values higher than the lowest pre-existing one, which is why priority values are chosen relatively large and closely-spaced.
It is guaranteed that no section type with a priority higher than the priority of the "id" section type will ever be introduced, so templates can always omit the section type descriptor for "context"- and "id"-sections, though omitting it for any other sections is generally unwise unless you (as the writer of the template) know that the set of extension specifications your implementation supports will never change.

To end this section of the spec, here is a graphic of the gender*render -template syntax described as a finite state machine (not taking into account the fact that not every section type is valid, and the rules about assigned section types and every type of section only existing once):



4.2 Pronoun Description Data

Pronouns description data, to which we will refer as gender*render -pronoun-data for the rest of this essay to spare us some words, is the way the user tells the render the pronouns of all people mentioned in a template so the renderer can render it. Any piece of text that fits the criteria described below is considered gender*render -pronoun-data, yet not every such piece of text necessarily works with every template, since it must provide the information required by the template for the rendering to work. Files that contain gender*render -pronoun-data should use the file extension `.grpd`.

gender*render -pronoun-data is a type of json data, which makes it easily parsable by any language.

To describe a single individual's pronouns (to whom we will refer to as individual pronoun data), a json object is used; several of these are then combined to provide pronoun data for all individuals. If a piece of individual pronoun data

is written into a file, the file extension `.idpd` should be used. Any json object whose properties are strings without whitespace, and whose items are strings, is syntactically valid individual pronoun data, though it might not be semantically valid, and even if it is, it might not work with every template depending on the information the template requires.

Analogously to what the concept of canonical context values and their aliases is for context values, individual pronoun data differentiates between specified properties (json properties for individual pronoun data that are hard-coded into this specification) and non-specified properties (those who are not hardcoded into the specification, yet are still valid). If, for example, "foo" is an alias for the attribute "bar", and a piece of individual pronoun data's "foo"-property has the value "foobar", then this individual pronoun data's "bar"-attribute's value is "foobar". Every specified property is either an *attribute*, or an alias for an attribute. The same goes for non-specified properties, contrarily to the behavior of context values, where not every non-specified context value is necessarily either canonical or an alias for a canonical one.

Every specified direct-mapped canonical context value is also a specified attribute when used as a property in individual pronoun data, and every alias of a specified direct-mapped canonical context value is also an alias for said attribute, when used as a property. Additionally, for every non-specified direct-mapped canonical context value ("custom property") "<*property*>", "<*property*>" is a non-specified individual pronoun data attribute, whilst "*property*" (assuming it is not already a specified property) and "*_property*" (with an `U+005F` underscore) are aliases of it.

When rendering a tag with a direct-mapped context value whose canonical context value is "foo", the tag simply gets replaced with the value of the "foo"-attribute of the individual pronoun data of the individual the tag refers to with its "id"-value; this behavior is explained in-depth later in the rendering section and subject to a view exceptions. A piece of individual pronoun data that lacks an attribute required by a specific template for a tag that refers to said piece of individual pronoun data is not valid for rendering this template.

Any piece of individual pronoun data that contains multiple properties that are aliases of the same attribute is always invalid (attributes count as their own aliases, for this purpose).

When it comes to describing individual pronoun data as the end user of a gender*render implementation, the recommendation is to use the attribute value itself, since it is the most descriptive of its aliases. If this is not an option for whatever reason, users can use the alias that doubles as the recommended alias of the corresponding context value (with the advantage of making the mapping between template and individual pronoun data more comprehensible), ore one of the other aliases, which are usually more descriptive whilst still being shorter

than the full attribute.

The following table gives you an overview over all attributes and properties that are not derived from direct-mapped context values according to the rules described above; the attribute in every list of properties is highlighted in italics. If one of these properties with a limited set of potential values has a value that is not allowed, the template is invalid.

information provided by attribute	attribute name and property name(s)	short explanation, where necessary
Gender-specific addressing	gender-addressing	If set to "false" or "f", the first name of an individual is used instead of its addressing; the only other valid values are "true" or "t". Defaults to true.
Gender-specific Noun handling	gender-nouns	Describes whether the person wants gender-specific nouns to use the female version where possible (e.g. "actress" instead of "actor"), the male version where possible (e.g. "fireman" instead of "firefighter"), or the gender-neutral version where possible (e.g. "firefighter"). Possible values for this property are `"female"`, `"male"` and `"neutral"`. Defaults to neutral.

Table 3: Supported properties in individual pronoun data

gender*render -pronoun-data is simply a json object whose properties are ids (strings without whitespace) corresponding to ids of gender*render -tags, and whose items are the individual pronoun data corresponding to their respective ids. Since the ids used by the gender*render -pronoun-data need to correspond to those used by the template, not every valid piece of gender*render -pronoun-data worked with every template. As specified later in the spec, renderers accept gender*render -pronoun-data as well as individual pronoun data in cases where no or only one id is used.

4.3 Pronoun Renderer

This section describes the way gender*render -specification conforming pronoun renderers work. It is divided into two subsections, one defining a gender*render -renderer, and one defining (additional) implementation guidelines that should be followed to ensure that all renderers use similar interfaces and users understand the renderer even if they used to work with a different implementation beforehand. The gender*render implementation that comes with this specification (`https://github.com/phseiff/gender-render`) also follows all of these guidelines.

4.3.1 gender*render Renderer specification

Any program that follows the specifications below is considered a gender*render -renderer. The purpose of such programs is to take gender*render -templates and gender*render -pronoun data and render them to texts that are gendered correctly according to the preferences voiced in the pronoun data. We will refer to gender*render -renderers simply as renderers for the rest of this section to aid the reading flow. An implementation that follows not only the main specification (this one), but also all extension specifications [6] may call itself a *full* gender*render implementation, or an *extended* gender*render implementation.

A renderer must take at least two inputs, a gender*render -template and a piece of pronoun data. As for the piece of pronoun data a renderer accepts, every renderer must accept gender*render -pronoun data as well as individual pronoun data, which is then processed into full gender*render -pronoun data following a number of steps explained below. The renderer may also be written in a way that allows to pass it a path to a `.gr`-file containing a gender*render -template instead of the template directly, or even in a way which exclusively allows this way of usage, though the later is not recommended and does not comply with the implementation suggestions given by this document. Analogously, the renderer may be written in a way that allows to pass it a path to a `.grpd`-file containing gender*render -pronoun data or a path to a `.gripd`-file containing individual pronoun data instead of the content of the gender*render -pronoun data directly, or even in a way which exclusively allows this way of usage, though the later is not recommended and does not comply with the implementation suggestions given by this document.

Along the rendering process, several errors might occur for several reasons. The way errors are classified and communicated to the user is not an implementation detail, but a part of the specification, since classifying errors is especially important due the logical, yet sometimes contra intuitive way gender*render renders templates. This specification thus defines not only what should raise an error, but also suggests different error names for different types of errors.

[6] refer to the *Specification Development* section to learn about extension specifications.

If the language of the implementation allows defining and raising custom error types, these error types must be defined and risen accordingly. If the language does not allow to define custom error types, yet allows to return information even if the execution of a program or function fails, the program or function must return information indicating what type of error occurred in a reasonable way. However, if the language of the implementation provides a standardized way to indicate a function failed to run, yet does not provide a way to return additional information about the cause of this failure, the implementation should use the standardized way of failing if an error occurs instead of returning information about the cause of failure.

The following types of errors are defined by this specification, and are used as described below. Please note that whilst the names of the errors are always written in camel case throughout this specification, the way they are written should be according to the official style guide of the language they are implemented in, if there is any. If the naming conventions of the language comply with the names of the errors defined in this specification, or if the language does not have any naming conventions, the names defined in this specification must be used.

Error name	commonly used for
`SyntaxError`	Used if the input is not a valid template and pronoun data aren't valid, independent of the way they relate to each other.
`InvalidCapitalization-Error`	Used if the capitalization of a context value or the capitalization value of a tag are invalid.
`InvalidPDError`	Used if the given pronoun data is not valid.
`IdResolutionError`	Used if matching individual pronoun data to tags does not wo out.
`MissingInformationError`	Used if the individual pronoun data a tag refers to does not contain the information the tag requires.
`DoubledInformationError`	Used if individual pronoun data defines a property multiple times with different names (possible since gender*render defines multiple different names for each property).
`InvalidInformationError`	Used if a property in a piece of individual pronoun data has a value it does not allow assigned.

Table 4: Types of errors that can occur whilst rendering

If the language of the implementation already has an error type of the name `SyntaxError`, and this error can be raised by the implementation manually, the implementation does not need to define a custom equivalent of this error type in their own namespace, and may instead use the pre-defined type. This is applicable for some languages like Python, and you can safely ignore it if it isn't applicable to your language of choice.
The errors (in languages where errors are objects) do not need to be defined as part of the global scope, if libraries or modules in this language commonly use their own scope (as is the case wit most common languages), and should default to the best practices for their respective language.
If the language is object-oriented, including custom errors, the errors defined by this specification may be derived from pre-existing error types, where fitting, as long as catching the exception based on the name defined by this specification is still possible.
Implementations in languages that support error hierarchies with the ability to find out whether a caught exception has been raised specifically, as opposed to being triggered by raising one of its derivates, may implement an arbitrary (as long as it is well-defined in their documentation) error hierarchy between its exceptions, based on good judgement[7].
Where possible, additional information regarding the cause of failure and how to fix it should be included when raising an error, but the way this is done is considered an implementation detail. Implementations should keep in mind that people using gender*render might not necessarily have read the spec, and might profit from self-explanatory detailed Tracebacks.

The rendering p-rocess uses different steps, described as follows. Please note that the order in which these steps are executed is not relevant; as long as the renderer is guaranteed to produce the same input-output-pairs as any render that accords to this definition does, it is up to the programmer how the renderer works internally. Each of these steps vaguely corresponds to one of the error types defined above, and raises almost exclusively said error if it happens to be unfinisheable.

The first step is parsing the input values (template and pronoun data) and checking them for correctness. If the received template is not a valid gender*render -template, a `SyntaxError` is risen; unless the issue is with the capitalization settings of a tag, in which case a `InvalidCapitalizationError` is raised according to the table above. If, on the other hand, the received pronoun data is neither valid gender*render -pronoun data nor individual pronoun data, an `InvalidPDError` is risen; this takes precedence over raising a `SyntaxError` in case both the template as well as the pronoun data are invalid. Please note that for a gender*render -template to be valid, not only does the syntax as describes via a formal grammar or a finite state machine be matched, but also does the determination of non-explicitly specified section types need work out,

[7] Refer to the *Error Hierarchy* section for additional recommendations.

as described in the template-part of this specification.

This step also contains unwrapping tags with multiple whitespace-separated context values, such as "{foo:bar * context:Mr_s Doe}", which are processed from a form along the lines of "{foo:bar * context:Mr_s} {foo:bar * context:Doe}" to two different tags separated by a single space (" ", U+0020). If a tags context value is not supported, an error may be risen, but this may differ from implementation to implementation to ensure backwards compatibility with versions that support a smaller set of context values.

After tags with multiple context values are split into sequences of multiple tags with one context value each, their capitalization value and their context value's capitalization are parsed and checked, taking into account the semantic sugar and other rules defined in regards to the capitalization-section (refer to the Template Language section for additional detail).

The second step matches gender*render -tags to individual pronoun data passed to the renderer. The crux of this is checking whether all ids used by the pronoun data match ids used by the template and vice versa, and making sense of individual pronoun data passed to the renderer. This step as well as the next one check not only whether the passed information are valid each on their own, but also whether they are matching. The procedures defined during this part of the specification walk a thin, yet clear, line between being too static and therefore forcing the user to provide not required information and reduce the ease of use of gender*render , and being to lash and therefore making debugging unnecessarily difficult. Understanding this part of the specification is crucial for using gender*render , and the information it gives should therefore be part of communicating the way gender*render can be used by implementation documentations.

The first part of this step is to deal with the fact that different amounts of id values can be used by different tags, and some tags don't have id values specified, and the given pronoun data might be individual pronoun data and therefore not specify any id values. To resolve this issue, the renderer assigns every gender*render -tag an id value if it doesn't have one already, and converts the given pronoun data to gender*render -pronoun data if it is individual pronoun data. The way this is done is described by the following table, which refers to the amount of id values specified by all gender*render -tags used in the given template as #ids:

	#ids = 0	**all tags have the same id (="bar") assigned**	**all tags have ids assigned, but not all the same**	**some tags have ids assigned, some not**

only individual pronoun data is given (=`idpd`)	set `pronoun_data = {"usr": idpd};` Assign id "usr" to every tag.	set `pronoun_data = {"bar": idpd}`	raise IdResolutionError	raise IdResolutionError
pronoun data is given for one id (="foo")	Assign id "foo" to every tag.	if "foo" $\neq$ "bar": raise IdResolutionError	raise IdResolutionError	raise IdResolutionError
pronoun data is given for n ($\geq$ 1) ids	raise IdResolutionError	if "bar" is not an id in the pronoun data: raise IdResolutionError	*nothing to do here*	if #ids + 1 $\neq$ n: raise IdResolutionError else: assign every tag without an id the id in the pronoun data that isn't assigned to any tag.

Table 5: Id resolution

After the instructions in this table are followed, every gender*render -tag in the template will have an id, and the given pronoun data will be converte to actual gender*render -pronoun information instead of potentially being individual pronoun data. The only thing left to do in this step is recreating the set of ids used by the template and the set of ids used by the pronoun data and raising an `IdResolutionError` if the ids in the template are not a subset of the ids in the pronoun data.

The third step does for the context-value of the tags what the second does for the id-value of the tags. It corresponds to the `Missing-/DoubledInformation-`error like the second step corresponds to the the `IdResolutionError`. This task is also finally the one that involves actually rendering the template.

Fist, the all individual pronoun data needs to be checked for doubled information. If any individual pronoun data uses multiple different property names to refer to one attribute, a `DoubledInformationError` is raised; see table 1 and 2 for information about which properties refer to which attribute.

Then, the renderer iterates over all tags in the template, and for each tag, the tags context value and the individual pronoun data provided for the tags id

value is taken, processed according to the following table, and the result then replaces the tag in the actual template. If an attribute is required for this yet not defined in the individual pronoun data, a `MissingInformationError` must be risen risen. Note that we will refer to attributes of the individual pronoun data simply as "attributes" in the following:

syntactic context of the gender*render -tag	procedure
Every direct-mapped context value with canonical *foo* except `address` and its aliases	The tag is replaced with the value of the *foo*-attribute.
Form of Address (context value with canonical `address`)	If the `gender-addressing`-attribute is set to false (or undefined), the tag is replaced with the value of the `personal-name`-attribute. Otherwise, it is handled like any other direct-mapped context value (see above).
Gender-specific Noun	See below.

Table 6: Rendering procedures for each syntactic context

As we can see, the only non-trivial case is correctly gendering gender-specific nouns, these being nouns that imply a specific natural gender of the person they refer to, or hyponyms for "person", to be precise. Depending on the value of the *Gender-specific Noun handling* attribute, such nouns, when given as the context value of a gender*render -tag, will be replaced by their correct equivalent for the specified gender. How these equivalents are determined is intentionally left vague since the english language is constantly shifting and unambigiously defining every rule for this would require to curate extensive lists. However, the recommended approach would be to use a graph of nouns that links every noun that refers to a person (for example based on WordNet[8]) to their synonyms according the gender implied by the synonyms. There are (not necessarily complete) datasets available[9] that do exactly this. It might be a good idea to test these sets on whether they contain a gender-neutral version for every gendered

[8]Princeton University "About WordNet." WordNet. Princeton University. 2010.

[9]for example `https://github.com/ecmonsen/gendered_words` on GitHub, which is an extension of the wordnet dataset, and is in turn extended by the data used by our reference implementation.

noun, and if they have only a version ending with "-man" and one version ending with "-women", to manually insert a version ending with "person".[10]

If a word is given as a gendered noun which isn't one according to the used data set, no error must be risen to ensure that every implementation terminates successfully for the same input data, but a (possibly suppressible) warning should be risen. If the implementation is also capable of determining if something is a word for a person a noun or a word at all, it may raise warnings for these as well.

After the string value a tag will be rendered to is determined, the capitalization option of the tag described by its `capitalization`-value is applied to said value by capitalizing it according to the capitalization-example provided in the capitalization value table above. If a character does not have both a lowercase- and an uppercase-version, it is interpreted as already capitalized correctly by this process (an important distinction because another way to implement this would be to skip the character and proceed with the next character as if there had not been any character in between). The whole process leading up to this, from the parsing of the template's context values and their semantic sugar to the capitalization of the final tag replacement, is referred to as the *global capitalization system* ("global" because it is indiscriminately applied to every tag at the end of its rendering process), a word which can be found all throughout the reference implementation.

After all of these steps are concluded, every gender*render -tag in the template will be replaced with a correctly gendered and correctly capitalized word for it. Afterwards, the rest of the template (save for the rendered tags) must be unescaped to remove single backslashes and replace double backslashes with single ones. If the implementation does not operate on a mutable string object (like they exist in some languages), the result should then be returned or outputted in another way, though this is not a must since some implications might not be encapsulated into their own function or program.

4.3.2 Implementation guidelines

In addition to the "must"s defined above, this sections lists some additional "should"s and guidelines for implementing gender*render renderers. This completed the previous section in that the previous section is about the way gender*render -implementations must work internally, whilst this section is about the way gender*render -implementation interfaces should be exposed to the outside, to encourage uniformity not only on the inside, but also on the outside. Renderers that follow all of these "should"s can call themselves conforming to the gender*render *implementation standard* (or, if they only follow the imple-

[10] This is what our reference implementation did with the data we used.

mentation standard for some of the extension specifications[11] they implement, specifically conforming to the implementation standards of the specifications whose implementation standards they follow).

In general, the purpose of this implementation standard is to define standardized interfaces and behaviors (function names, parameter names, additional behaviors and parameters, optional optimizations, test functions for the user to find potential caveats in their templates etc.) so that every user who is comfortable with one implementation of gender*render is capable of using any implementation in any other language without having to refer to the documentation to get started. It is acceptable that not every implementaon might be able to follow these guidelines, though, and they are admittedly optimized for object-oriented languages, though they also contains alternate suggestions for non-object oriented languages.

In addition to being a guideline for other implementations, this standard served as a basic outline for writing the exemplary implementation that comes with this specification. Whilst said implementation strictly follows all of these guidelines, it should be mentioned that was written after the standard, so these guidelines are in no way based on the original implementation or, even worse, written to justify this implementations design choices.

The terms "suggestions" and "guidelines" are used interchangeably here.

4.3.2.1 Naming Guidelines

All naming suggestions given in this section follow Python coding conventions, these being `CamelCase` for classes, `snake_case` for variables, functions, arguments and methods, and `kebab-case` for package names. Whilst the names are part of these suggestions, their case is not, and every implementation should follow the naming conventions of its respective language depending on the type of things the names refer to.

4.3.2.2 Object Orientation

The guidelines are mainly targeted towards object-oriented languages, since most high-level languages are object oriented and there is not much reason to implement gender*render using a low-level language. However, they can and should be applied to non-object oriented languages as well, following the following table of analogies; in cases where the left-sided concept is available for use, it should be used, otherwise, the right-sided concept should replace it:

object-oriented term	not object-oriented equivalent

[11] refer to the *Specification Development* section to learn about extension specifications.

object representing a foo	data structure representing a foo
class constructor	function that returns the data structure that compensates for the object
method of an object representing a foo; takes e.g. a, b, c as arguments	function that manipulates a data structure representing a foo or returns a manipulated version of it (depending on whether it is mutable or not); takes e.g. the data structure, a, b, c as arguments
function that mutates foo	function that returns a modified copy of foo
function `f`, which accepts some optional arguments	function `f_ext` where all those optional arguments are required, and function `f`, which does not require the optional arguments and internally calls `f_ext` with the default values for the omitted arguments; unless there is a standard way to deal with issues like this in the given language, which should then be preferred to this approach.
boolean type	0 or 1 as integers

Table 7: OOP equivalents for low-level languages

4.3.2.3 Naming the package

Packages (as in, modules or libraries that can be installed via a package manager, be it language specific or general) should be called "`gender-render`" if they target the english language. If they target a different language, they should be extended with a hyphen and the corresponding language code (e.g. "`gender-render-de`"). In case the implementation is highly language specific (e.g. intended to be used from within one specific language rather than as a tool via the command line), and the package manager it is uploaded to is not language-specific (unlike e.g. nvm or pypi), the package name should be extended with a hyphen and the common file extension of the language (e.g. "`gender-render-js`" or "`gender-render-de-js`"). In cases where other packages of the same name already exist, a name should be chosen with good judgement.

4.3.2.4 Naming the module/ library

Most languages support writing modules and libraries that can be imported, embedded or otherwise injected into any piece of code usg a simple statement that contains the name of the module somewhere in it (e.g. "`import`

gender-render" or "#include <gender_render>"). Whilst some languages come with their own packet manager, some don't, and some that do support using different names for the package and the module that it installs. Therefore, a separate naming convention for module names is necessary.

Module names are formed like package names, except without the appendix for the language name, and using the case conventions for module names rather than package names. In cases where the package name differs from the scheme above, the module name should be based on the package name, but also without the additional information regarding the implementation language.

4.3.2.5 Core functionality

This is the functionality that every gender*render implementation should expose, if appropriate (within good judgement) its main namespace.

class `Template`**:** A class representing a parsed, but not yet rendered, gender*render -template. Its constructor takes the template-related information that any gender*render -renderer takes, parses and processes it as far as possible, and an additional `render`-method accepts the pronoun-related information that any gender*render -renderer takes and renders both, with the result then being returned. This practice of implementing an object or data structure just for processing the template and separating this process from the actual render is motivated by the need for increased performance in case of multiple subsequent renders of one identical template.
The constructor accepts one positional argument, "`template`", which accepts a string, and one optional argument "`takes_file_path`", which defaults to false and accepts a boolean. If `takes_file_path is true`, the `template`-argument is interpreted as a file path to a `.gr`-file containing the template; otherwise, it is interpreted as a string containing the template.

method `render()`**:** A method of the aforementioned `Template`-object. It accepts a piece of pronoun data and returns the rendered template. Accepted arguments are "`pronoun_data`", which accepts a string, and one optional argument "`takes_file_path`", which defaults to false and accepts a boolean. If `takes_file_path is true`, the `pronoun_data`-argument is interpreted as a file path to a `.grpd`- or `.gripd`-file containing the pronoun data; otherwise, it is interpreted as a string containing the pronoun data. Implementations may write `render` as a method that accepts a JSON-like object (like a `dict` in Python or an object in Javascript) if it receives one instead of a string containing json.

function `render_template()`**:** `render_template()` is a shorthand for `Template().render()`. Order in which arguments are passed to `render_plate()` is first required arguments for Template(), then required arguments for render(), then

optional arguments shared by Template(), then optional arguments for Template() and then optional arguments for render().

class `PronounData`**:** A class representing a parsed piece of individual pronoun data, with all calculations that can be done before knowing which template it will be inserted in already done. This type of object is accepted by every function which accepts pronoun data, such as `Template().render()` or `render_template()` wherever these functions accept a piece of pronoun data (the `pronoun _data`-argument). Its constructor accepts the same arguments as the `render()`-method as described in the preceding sections of this list.

4.3.2.6 Warnings

As in every program, there are multiple scenarios that can arise along the render process that might warrant raising a warning. This specification does not define any warnings that must be risen to comply with these standards; rather, it suggests some warnings that should be implemented if the architecture of the program warrants it, was well as guidelines on how to handle the disabling and enabling of warnings.

Every warning suggested by this section comes with a name that should, like every other name specified here, adjusted to the styleguides of the implementation language. All of the names of these warnings end on "Warning" as a uniting factor in their names, but this uniting factor should be left out or replaced according to the coding guidelines of the respective language. If, for example, the suggested name is "`FooBarWarning`", but the language and/or coding guidelines used for the implementation require warnings to be snake_case and to start with "potential_problem̈rather than ending with "Warning", the warning should be "`potential_problem_foo_bar`".

Which function should raise the warning is implementation dependent; however, every function suggested by this specification should have an `warning_settings`-argument which takes a value defining which warnings should be enabled and which warnings should be enabled for the run. The syntax and type of this warning is up to the implementation; however, it should allow to individually enable and disable every warning, and there should be constant values for setting all warnings as enabled/ all warnings as disabled defined. If the implementation of this feature is string-based, e.g. the warning settings get passed as an array of strings describing warning types, the string names representing the warnings should be interpreted according to this specification and in snake_case, so "`FooBarWarning`" would be represented by "`foo_bar_warning`". This ensures compatibility of values for the `warning_settings`-argument across all implementations of gender*render . The default value for the `warning_settings` argument is up to the implementation, but should be chosen to comply with the warning sensitivity common in programs written in the respective language.

The value of the argument should be passed from every function to every function it calls (assuming both functions are part of the program and have potential warnings to raise). `render_template()`, for example, should pass the user's warning preferences on to `Template()` as well as its `.render()` method.

warning name	warning meaning
`NotAWordWarning`	The value of a Gendered Noun-tag is not a word known in the english (or implementation-specific) language. Raising this warning only makes sense for implementations that come with a full dict of the english language.
`NotANounWarning`	The same as above, but specifically for valid words that are not nouns.
`NotAPersonNoun-Warning`	The same as above, but specifically for valid nouns that do not refer to a type of person or profession and are thus not gendered in any way.
`FreeUngendered-PersonNoun-Warning`	A noun that refers to a type of person or profession was found outside of any tag.
`FreeGendered-PersonNoun-Warning`	The same as above, but specifically for nouns for whom more than a single, gender-neutral form exists.
`UnknownProperty-Warning`	A custom property in a piece of individual pronoun data is not using the special syntax for custom properties ("*_property_name*" or "<*property_name*>" rather than just "*property_name*"). Using the special syntax is preferable to ensure that the custom property is not named equal to a property introduced in a later version or addition to this specification.
`FreePronoun-Warning`	A (non-neo) pronoun is found freely outside any gender*render -tag.
`UnexpectedFile-FormatWarning`	A file name specified to Template() or Template().render() does not follow the file extension naming convention of this specification.
`IdMatching-NecessaryWarning`	The set of ids specified in the given pronoun data was not equal to the set of ids used in the given template, but could still be matched. Since this might happen on accident, a warning should be present.

Table 8: Potential Warnings

Not all of these warnings need to be implemented, and an arbitrary amount of additional warnings may be implemented in addition to them. Warnings

should come with the information required to track them down to their cause (the rerence implementation might set a good example here).

The recommended approach to warnings about the presence of ungendered words of any form (these warnings all start with "`FreeUngendered`" in the table above) would be to split the text surrounding tags into words, with non-letter symbols serving as separators, and checking every word for presence in a dedicated list or other data structure of gendered words. Since not all letters are part of ASCII code, checking whether a character is part of the alphabet is not enough to check whether it is a letter in many languages; implementations should instead check whether a character is a "Letter" according to the Unicode standard to determine whether it is part of a word or a separator between words[12].

In addition to the warnings defined by this section of the specification, implementations may implement an arbitrary amount of additional warnings to give even more fine-grained feedback during the parsing and rendering process. These additional warnings can be disabled by default or enabled by default, depending on design decisions made by the implementation; the ability to suppress warnings that comes with gender*render makes both decisions valid.

The following is a list of all additional warnings the gender*render reference implementation uses (or used in previous versions) in addition to the ones specified above; they where never part of the specification, though, since their usefulness can be debated and/or is highly language dependant. The reason why there is still a section in this specification for them is because it might be of interest for other implementations to see which additional warnings where implemented in the reference implementation and why (if ever) they where removed:

warning name	warning meaning (slightly modified from the docstring)	reason for removal

[12] Compare to how the `str.isalpha()`-method works in the Python programming language, for example.

`DefaultValue-UsedWarning`	An attribute with a default value was looked up in a piece of individual pronoun data, but not found, so its default value was used. This is in itself not a problem and perfectly fine behavior; this warning is only raised to inform the user in case they forgot to define the property or pythonically prefer explicit to implicit.	N/A
`GenderedNouns-BuildFromWeb-Warning`	The data containing the gendered and especially neutral versions of all english hyponyms for "person" could not be found; therefore, it will be downloaded and saved from the internet. This should only happen once per installation and only when initializing the module for the first time, and it should not happen at all with the PyPi installation.	N/A
`NounGendering-GuessingsWarning`	Raised if a noun will be gendered, but based on automated guesses at the correctly gendered version of the noun rather than hardcoded values. This warning exists because the data used by the implementation to find the correctly gendered version of every word had holes, missing information and contradictions in the hundreds, all of whom where fixed by an algorithm and not yet manually checked for correctness.	N/A

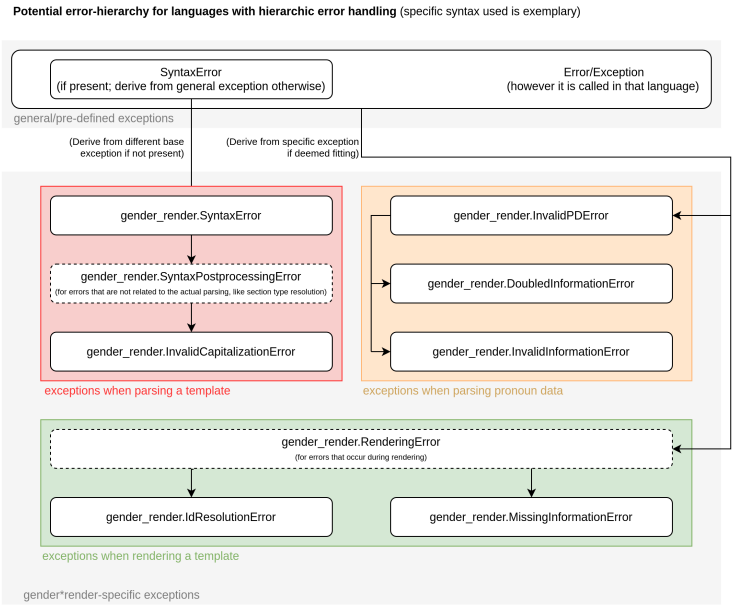
Table 9: Additional Warnings in the reference implementation, and why they where removed

4.3.2.7 Error Hierarchy

Some languages (mainly object-oriented languages such as Python and Java) have error hierarchies, where Error- and Exceptions Types can have sub-types, and catching an Exception catches all of its sub-exceptions as well (usually, that means that Errors and Exceptions are implemented as classes that just inherit from each other). For this reason, implementations in such languages (like the reference implementation) might want to define hierarchies between the gender*render exceptions they define. This is in accordance to this specification, which does not forbid such behavior. Implementations may also add additional exceptions into their error hierarchies, and decide freely from which pre-existing exception they derive the exceptions defined by this specification; the only limitation is that catching an exception defined by this specification must always catch every issue that, as fined by this specification, raises said exception.

Some languages might not allow to check whether a caught exception is exactly the exception one tried to catch or an exception derived from this exception (so catching exception A does not allow one to determine whether exception A was risen or exception B when B is derived from A). Implementations in languages like this may not derive exceptions defined by this specification from other exceptions defined by this specification, since e.g. deriving `DoubledInformationError` from `InvalidPDError` in such a language would make it impossible to specifically catch `InvalidPDError`.

The following graphic shows the (exemplary) error hierarchy used by the reference implementation, which might serve as a good example. Arrows go from parent exception to child exception, and exceptions in dashed lines are exceptions that the implementation added on top of the set of specified exceptions to allow for even more fine-grained error handling. Ultimately, the hierarchy between different exceptions is opinionated and the ideal error hierarchy might vary depending on the application area and language of the implementation, but the following hierarchy might still be of worth and work quite well for many implementations that qualify for using an error hierarchy:



5 Specification developement

This section lays out some details of the way this extension is versioned and may be extended or developed in the future.

As it is now, this specification is nowhere near its full potential, yet it is usable and quite enough to be a valid proof of concept. I consider it to be enough to publish and share, but there will be imperfections or maybe even straight out design faults. Thus, issues and suggestions are very welcome in the GitHub repository where this specification is developed[13].

Future development will go two paths in parallel: One path is the steady improvements of this specification in wording as well, if necessary, content. The other path is releasing *extension specifications*, these being additional specifications that introduce new rendering steps and attributes as well as further implementation suggestions to enhance gender*render implementations with useful extra functions. The extension specifications will not extend the specification in that every implementation must follow each extension spec to be considered compliant with the main specification, but rather in that every implementation must follow the "must"s defined by this specification and may follow an arbitrary amount of additional extension specifications. An implementation that follows extension specs Foo and Bar in addition to the main spec is therefore

[13] `https://github.com/phseiff/gender-render/issues`

not more compliant with the main spec than any other implementation, but it can say of itself that it implements extension specs foo and bar in addition to the main spec, like a program coming with an add-on pre-installed; additional definitions on how to determine whether an implementation follows this standard or not can be found at the beginning of both subsections of the *Pronoun Renderer* section of this specification. The idea behind this is to be able to rapidly introduce new features without every implementation constantly having to adapt to these new features to ensure it doesn't become obsolete, and to allow a more organic development of gender*render driven by public response to its components rather than one person scribbling down their visions into a spec that becomes more and more bloated in the process.

In general, the amount of work and time I plan to put into maintenance and extension of this specification largely depends on the amount of reception it receives; however, issues will definitely be addressed as in any maintained project. Please also note that this specification is developed in the same repository as the exemplary implementation it comes with; but that its development does not depend on said implementation.

This specification uses a variation of semantic versioning, in which the major version increases through each backwards-incompatible change as per usual (this will, however, not happen once the project enters a staple state or after it establishes itself, if it does), the minor version with each feature added (which should usually be done via extension specs, though it might be necessary to merge extension specs into the main spec if it turns out to lack required functionality) and the "bugfix" version with every change that does not affect its content, but its content's wording.

The current versions major number is 0, since there might be need changes according to the feedback it receives. For as long as the major number remains zero and no notable adaption of the standard can be seen, minor breaking changes may occur without increasing the major version number.

Recent versions as well as their changelog will always be accessible via the specification download page of the project's GitHub repository[14], which is always where the latest version, future versions and extension specifications are and will be located.

6 Exemplary Implementation

To illustrate the specification and approach it in a practical way as (1) the practical side of this proof of concept and (2) to ensure that there is a useful

[14] `https://github.com/phseiff/gender-render/#download-specifications--changelog`

implementation that makes the spec reality, the specification comes with an exemplary implementation written in Python. This implementation is written according to the specification, not the other way around, not even chronologically. The example implementation also follows and demonstrates all the interface and implementation suggestions of this specification, and is well-equipped with automated testing.

Its documentation as well as installation instructions can be found in the GitHub repository of this project.

7 Outlook

As our world as well as our perspective on society and gender evolves, queer identities and people who openly identify as non-binary genders will become far more common occurrence than they already are. As non-binary people will become less and less ignorable in politics, their ignorability in technology will decrease as well, and it is part of our responsibility as developers to not fail in following these social developments and overcome technical challenges they bring with them like any social development does. Developers and technicians have overcome an incredible amount of challenges and technical limitations in and of the past three decades, many of whom where caused by real-live developments and the natural striving of humans for self-fulfillment. It would be a disgrace to IT if we failed to keep up with social developments and proactively slowed them down for technical pretexts, and IT refusing to find solutions for correctly gendering *every person* in *every* automated text because of technological pretexts really puts a bad light on us; after all, our profession was involved in getting humans to the moon before we even knew we'd ever send an automated email, so how can we fail at making people feel accepted when we made people literally reach for the stars?

This projects goal is by no means to create the one only true way of automated email gendering, but rather to address an issue and demonstrate that solving it would not require sorcery. There will probably be a lot of projects, programs and specification to address the issue of correct automated gendering in the future, and I do not expect this specification to still be around when these take off. This project will hopefully sensibility some people for the issue, show them that and how it can be addressed, and it will hopefully only be the first step in a long line of similar projects great enough to make this one become forgotten.

8 License

This version of the specification is licensed under OWFa 1.0 by phseiff (contact: phseiff@phseiff.com).