Unicode MessageFormat 2.0

The “EM” Proposal Draft, February 2022

Champions:

- Mihai Nita (Google)
- Elango Cheran (Google)

Introduction

For more than two years the message format workgroup worked on a proposal for a new message format version.

The new version should be able to support more advanced features than the current solutions, and can become a universal solution for localization, regardless of programming language, operating system, framework, etc.

The initial demand was triggered by ECMAScript, which has no standard. But we identified it as a bigger opportunity, because in fact there is no standard that works well across platforms and programming languages and is widely adopted.

The ICU `MessageFormat` is standard and well established, but the adoption is not as widespread as we would like, particularly the support from standard localization tools.

Here are some early documents created and agreed upon by the workgroup participants: *Why MessageFormat needs a successor* and *Goals and Non-Goals*.

The following are some of the limitations of ICU's MessageFormat that need to be addressed:

- **The list of formatters and selectors is fixed in the spec.**
  That currently consists of `number`, `date`, `time`, `spellout`, `ordinal`, `duration` for formatting, and `plural`, `selectordinal`, `select`, `choice` for selection.
  And because it didn’t change often, many tools don’t expect it to ever change, and would break if faced with something new. This makes any enhancements difficult.
- **The format only represents info for runtime rendering.**
  There is no standard way to pass info for translation: comments, rules, links to screenshots, examples, etc.
- **There is no way to represent complex grammatical concepts like agreement and inflections.**
  For example grammatical agreement, or a translator specifying the gender / number / case of a placeholder.

- **It is designed for plain text.**
  Anything that needs formatting (like HTML tags) is “a hack” on top of it.
- **Certain features make some bad i18n practices too easy.**
  For example using patterns in the datetime placeholders.

Proposal

The work group agreed to start by creating a data model of the message, which is a language and syntax independent representation.

This gives us some independence to discuss functionality requirements separately from the exact syntax details, and it would allow us to create different yet functionally-equivalent syntaxes that are suited to different technologies.

And if the model has a superset of the functionality of the existing syntaxes (ICU MessageFormat, Fluent, FBT), it might also be possible to parse legacy formats into the same in-memory data model, which would allow backward compatibility, migration, refactoring, etc.

The other big decision was to create extension points that allow the standard to grow without breaking.
The idea is similar to the way BCP 47 works in conjunction with the IANA Language Subtag Registry.

This is of course not final. But it is the main outline, the house frame on which we add the final details.

Guiding values

For any complex endeavor it is useful to explicitly state what we value most when we make our decisions.

- **Pragmatism** - We believe in solving real problems faced by internationalization developers, translators, localization tooling, and other stakeholders.
- **Scale** - The solution must also be able to work for anything from small to large projects, which may have many components and heterogeneous tech stacks. This is a need faced by many companies, from big to small. Also, translators working in high volumes need simple designs, and cannot be expected to have technical skills.
- **Adoption** - A standard, no matter how perfect, does not have much value if it is not adopted.
- **Localization integration** - The solution must be compatible with existing localization tooling and workflows, and a deliverable to that end is a standard mapping to/from XLIFF 2.x. But it should be done in a way that does not require changes in the core assumptions many localization tools have. It is not sufficient to merely create a custom

XLIFF extension that is not standardized, when even standardization does not guarantee adoption.

- **Developer adoption** - It should be possible to migrate messages from an old syntax to the new one (ideally automatically), without losing functionality. The messages should be able to exist independently of each other such that migrations can be incremental, not tied together such that a migration becomes an “all or nothing” proposition.
- **Prefer simplicity for translators when necessary** - A design that is simple enough to support the needs of translators can also be used by developers with more complex needs, but not necessarily vice versa. If a conflict in needs arises, we give priority to translator / localization needs, as what we design gets used as a localization format, in practice.

Details

The big picture

The data model

The structure below represents an internal data model.
This can be represented with different surface syntaxes, based on the environment.

The model is simplified for presentation here, and to see the “*Details, examples, and extras*” section for details.

Various elements (messages, placeholders) will have other “pieces of info” attached (flags, metadata, etc). But are not shown in this overview for better readability.

```
Message ::= Macros* (SimpleMessage | SelectorMessage)

SimpleMessage ::= PlainText (Placeholder PlainText)*

PlainText ::= <code-point>*

Placeholder ::= ArgNameOrConst flags*
              | ArgNameOrConst flags* formatFunctionName
              | ArgNameOrConst flags* formatFunctionName formatOptions

ArgNameOrConst ::= argumentName | constant
argumentName ::= string
constant ::= string

flags* ::= ':open' | ':close' | ':standalone'
```

```
FormatOptions ::= map<String, ?>

// ordered for round-trip syntax->model->syntax,
// not necessarily for runtime selection
SelectorMessage : SelectorArg+ OrderedMap<SelectorVal+, SimpleMessage>

SelectorArg : varName
            | varName selectFunctionName
            | varName selectFunctionName selectOptions

SelectorVal : number | string

SelectOptions : map<string, ?>

Macro ::= no format description yet (to reduce verbosity)
   But they are a simplified Message.
   Without Macros (to eliminate recursivity),
   TBD what else to disallow (for example multiple selectors?

The '?' in FormatOptions / SelectOptions is at least a string.
   TBD if it needs to be more than that (numbers, booleans)
```

The registry

One of the limitations we've seen with `MessageFormat`, and something that other standards also identified, is that hard-coded lists of acceptable values are difficult to change later on.

By contrast, many standards have a "core" that is hard to change, and a "registry" that can be easily updated.

Examples of this approach:

- BCP 47 + IANA Language Subtag Registry + extensions (think `-u-` and `-t-`)
- Unicode Variation Selectors + Ideographic Variation Database
- XLIFF 2.x and extension modules
- XML spec + DTDs / Schemas

For our `MessageFormat`-like functionality we need two kinds of functions:

- Formatting functions. Similar to ICU's number, date, time, duration formatters.
  They take a value and return a "formatted fragment" that can be inserted into the rest of

the message. The fragment does not have to be text, might contain structure, for example TTS info, or semantic info about ranges (think `formatToParts`)

- Selector functions (similar to today's plural, selectordinal, select)
  They are used to choose the best match between several message variants based on some input conditions.

The signatures of these kinds of functions will be part of the standard, but the exact list of functions will not.

Unicode would maintain a registry, with the initial version of the registry containing the existing functions.

There can also be company / product specific registries that can be passed to localization tools, so that they know what is allowed to change and what is not. Similar to XML namespaces, or the `-x-` extension in BCP 47.

A function entry in the registry would contain the function name (for example "datetime" or "plural"), the parameters ("skeleton" or "day" / "month" / "year" / "timezone" etc (like the ECMAScript option-bag)).

Sample syntax and examples

The following represents how the data model would appear in a syntax that is a natural extension of the ICU syntax. The EBNF is simplified for review; for example, the places where spaces are allowed is not explicit.

The syntax used exists mostly so that we can have something readable to look at, but it is not final, could change as we go.

Simple messages

Simple messages are a linear sequence of "parts" that are plain text or placeholders.

Some elements are not presented in this simplified model for clarity. Some examples: flags indicating the display context (sentence case / standalone / etc), post-processing directives (to "fix" matching vowels, etc).

The plain text can be translated, and is rendered as-is at runtime.

Placeholders can't be translated, but they can be moved around, optionally added / deleted (based on flags that control that), and **have attributes that might be changed by translators**.

The placeholders have the name of the item to format, the formatting function name (string), and parameters that control how the formatting happens.

Translators can change some parameters, but not all (function dependent).

Below are some simple examples, with a few more complex ones in the "*Placeholders: using and extending*" section.

A simple example:

```
"Hello {$user}!"
"Hello {$user person_name}!"
"Hello {$user person_name style:formal}!"
```

Grammatical features:

```
"I gave a book to {$name grammar case:dative}!"
"I've lost {$item grammar article:definite case:accusative}"
```

The order of the options does not matter. We think of these options as named parameters for a function (or an unordered map, or an "option bag" in JavaScript).

Various ways to control the formatting (options):

```
"Your card expires on {$expDate datetime style:long}"
"Your card expires on {$expDate datetime skeleton:yMMMMd}"
"Your card expires on {$expDate datetime year:numeric month:long
day:numeric}"
```

And if new ways to specify the format are developed, they are added to the registry without disrupting the core standard.

Selector messages

The model:

```
SelectorMessage : SelectorArg+ Map<SelectorVal+, SimpleMessage>

SelectorArg : varName
            | varName selectFunctionName
            | varName selectFunctionName selectOptions

SelectorVal : number | string

SelectOptions : map<String, ?>
```

Let's start with a simple selection the way it is currently done in ICU (and very similar in Fluent), here in ICU syntax for illustration, not yet in proposed MF2 syntax:

```
{count, plural, offset:1
     =1 {{guestName} left}
     =2 {{guestName} and one other guest left}
  other {{guestName} and # other guests left}
}
```

The `count, plural, offset:1` is the same thing as a `SelectorArg`.

It the [ICU MessageFormat documentation](), these are named:
`simpleArg = '{' argNameOrNumber ',' argType [',' argStyle] '}'`

For example, in ICU MessageFormat, we might describe the equivalent of a selector called `"plural"` as:
`pluralArg = '{' argNameOrNumber ',' "plural" ',' pluralStyle '}'`

The new selector message is very similar, but instead of a single selection rule and values, it uses a sequence of selectors and sequences of values.

Example of a selector message:

```
{ [ {$count plural} {$host_gender gender} ]
    [    =1 female] {One guest to her party} // simpleMessage 1F
    [    =1   male] {One guest to his party} // simpleMessage 1M
    [    =1  other] {One guest to their party} // simpleMessage 1O
    [other female] {{$count} guests to her party} // simpleMessage NF
    [other   male] {{$count} guests to his party} // simpleMessage NM
    [other  other] {{$count} guests to their party} // simpleMessage NO
}
```

The approximate EBNF:

```
SelectorMessage := SelectorArg+ SelectionMap

SelectionMap = SelectionEntry+

SelectionEntry = SelectionMatch Submessage

SelectionMatch= "[" SelectorVal+ "]"

Submessage = "{" SimpleMessage "}"

// each SelectionMatch has the same item count as the SelectorArg count
// but we can't express this in EBNF
```

The colors show the correspondence with the data model:

```
SelectorMessage ::= SelectorArg+ Map<SelectorVal+, SimpleMessage>
```

It is very similar to the previous approach, but the selection part and the matches are now tuples, instead of single values.

This eliminates the deep nesting problems that existed before, and the proposed model and syntax also force the use of complete messages for each case.

Multi-selector matching algorithm

Even with the current implementation, the selection is not simply:

```
tmp = selection_function ( input_value )
find the entry that matches tmp
```

Because if we look at `plural`, an input value 1 matches several entries: `=1`, `one`, and `other`. And the runtime should select the best match, which is `=1`, regardless of the order of the entries.

So the `selection_function` should not take the input value and return a keyword.
It should take the input value plus all the existing keys, and return the best match.

For this selection message example containing one selector (ICU syntax):

```
{$count, plural,
     =0 {...}
     =1 {...}
    one {...}
  other {...}
}
```

When `count = 1`, the selection function can't simply do this:
`plural(1) returns one`
Instead, it must do this:
`plural(1, [=0, =1, one, other]) return =1`

If `=1` is missing, then it would return `one`.

So the current selection behavior is a best match strategy, not a first match strategy.

How do we extend from single to multiple selectors?

We can extend a selection message to have multiple selectors by converting the selecting condition at the top to be an array of conditions, and extend the matching keywords for each message to be a corresponding array of keywords:

```
{ [ {$bookCount plural} {$ownerGender gender} ]
  [      =1                female ] { This is her book }
  [      =1                  male ] { This is his book }
  [      =1                 other ] { This is their book }
  [   other                female ] { This are her {$bookCount} books }
  [   other                  male ] { This are his {$bookCount} books }
  [   other                 other ] { This are their {$bookCount} books }
}
```

The one problem left to solve is how to select from these multiple options with multiple conditions.

When all the combinations of cases are present in the message, an exact match works well.

But we need an algorithm to select the best match when we have an incomplete set. For example:

```
{ [ {$bookCount plural} {$ownerGender gender} ]
  [      =1                female ] { ... }
  [      =2                 other ] { ... }
  [     one                 other ] { ... }
  [     one                  male ] { ... }
  [   other                 other ] { ... }
}
```

How do we select the best match for `{itemCount:1, itemGender:male}`, which does not exist in this exact combination?

We have several proposals in Annex 2: Multi-selector matching algorithm alternatives.

Multi-selector syntax

We used the above multi-selector syntax in order to keep things close to ICU for illustration purposes, we use `[ ... ]` to represent the idea of “an array of items”, and we wrapped the selector function with its style in `{}`, because this is what ICU uses for arguments.

There is no technical reason for an ICU-inspired syntax other than familiarity.

But since the result is not backward compatible with ICU anyway, we can afford to tinker with the syntax to make it less “noisy”.

And when reduced to a single selection, the ratio of useful info to noise is even lower:

```
{ [ {$count plural offset:1} ]
    [   =1] {{guestName} left}
    [   =2] {{guestName} and one other guest left}
```

```
    [other] {{guestName} and {#count} other guests left}
}
```

So in Annex 3: Alternate syntaxes for multi-selectors we examine a few alternate syntaxes that are easier to read and write by a human, while still carrying enough information to parse without being ambiguous.

Selection depending on a formatter result

We will present this with a single selection, instead of tuples, for clarity. But it would extend to tuples.

Example:

```
{ [ {$amount plural} ]
    [   =1] {{$amount} dollar}
    [other] {{$amount} dollars}
}
```

But the plural selector can't decide what the best match is just by looking at the input value. It also depends on how the value is formatted. For the same input value (1) we say *"1 dollar"* (`=1`) but *"1.00 dollars"* (`other`).

Worse, we can have styles in the placeholders that can lead to inconsistent results:

```
{ [ {$amount plural} ]
    [   =1] {{$amount number trailingZeroDisplay:HIDE_IF_WHOLE} dollar}
    [other] {{$amount number integer} dollars}
}
```

So we need the selector function (`plural`, `selectordinal`, etc., and in the future: `gender`, `case`, and more) to somehow be aware of what the formatting function does.

Another example:

```
{ [ {$range select} ]
   [same] {We are closed on {$range interval_format skeleton:yMMMd}}
   [diff] {We are closed between {$range interval_format skeleton:yMMMd}}
}
```

The result depends on the input, formatting function, and argument style.

The select function can't just test `range.start == range.end` to decide, because there might be hour differences. And hour differences are only visible if the skeleton specifies an hour.
So the `select` function would need to look at the argument style of `interval_format`, and also to know what `interval_format` does with that style.

Allow for selector functions that combine formatting and selection:

```
{ [ ($amount plural_number trailingZeroDisplay:HIDE_IF_WHOLE) ]
    [   =1] {{#amount} dollar}
    [other] {{#amount} dollars}
}

{ [ ($amount plural_number trailingZeroDisplay:HIDE_IF_WHOLE signDisplay:always)
    ($count plural offset:1) ]
   [    =1    =1] {{#amount} dollar for {$firstFriend}}
   [    =1   one] {{#amount} dollar for {$firstFriend} and {#count} other}
   [    =1 other] {{#amount} dollar for {$firstFriend} and {#count} others}
   [ other    =1] {{#amount} dollars for {$firstFriend}}
   [ other   one] {{#amount} dollars for {$firstFriend} and {#count} other}
   [ other other] {{#amount} dollars for {$firstFriend} and {#count} others}
}
```

With this, the formatting options are shared across the selection, making them consistent and avoid their duplication. It would be discouraged (lint?) to use an {$amount} instead of the # and thus with different formatting options.

So the code that implements the “rendering” of a MessageFormat would invoke the `plural_number` function that would do both formatting and selection.

This can increase the number of functions a developer needs to implement (ex: `plural_number`, `plural_duration`, `plural_interval`, `gender_person`, `gender_date`, and so on).

But in all these cases, the formatting functions need to know what the requirements are for the selector functions that will take its formatted value as an input. This shows that there is a strong coupling between formatting and selection, even if we try to separate them (see Annex 4: Alternative for selection depending on a formatter result).

A distance formatter would need to return information needed by selection functions for plural, and gender, and maybe case.

So we might as well make this clear and combine the functionality into one function doing both formatting and selection.

An alternative we considered was to decouple the formatter and selector function, and pass some information between them. But it does not really reduce coupling. In fact, it can be worse in some cases. The formatting function would need to return information for all kinds of selectors, when in reality it would only be used with one.

In summary

This current proposal does not offer an algorithm, although several were considered.

But these would would be the proposed principles for a selection algorithm:

- Must be predictable (easily) for translators and developers
- Must be robust in the face of additional values
  (e.g., adding “many” for French doesn’t make old messages produce horrible results)
- Ideally the “selection entries” (SelectionEntry+) would be unordered.

Placeholders: using and extending

Extra information needed

Since the placeholders can use registered functions, they can be very flexible, without being “free form.”

All placeholders will have extra information (inspired by XLIFF) that will tell us what can be done with the placeholder. This was left out of the main “big picture” data model for clarity.

A very useful property of a tag is to know if it is `open`, `close`, or `standalone`. The default is `standalone`.

Explaining `open` / `close` / `standalone`

The current `MessageFormat` allows parts of the message to be replaced at runtime with formatted (converted to text) values. Although ICU calls them “arguments”, we use the localization industry’s term “placeholders”.

In modern software, UIs allow for formatting attributes that previously were possible only in documents (links, bold, italic, color, font sizes, bullets, etc.)

So it is now very useful to be able to represent such non-textual markup concepts in our syntax, otherwise they will be implemented in custom (and non-portable) ways.

And standard localization tools will not understand such custom, non-standard markup and therefore not be able to validate them.

Currently, most localization tools can handle these concepts, because they have had to deal with them for a long time. Many localization tools support files like HTML, Markdown, MS Word and Powerpoint, OpenOffice, InDesign, FrameMaker, templating languages.

We should incorporate the wisdom of their widely-used design, which is the product of several iterations to solve problems encountered in their experience.

The idea is simple if we look at HTML and XLIFF: some tags should be in pairs, open-and-close (for example `bold`, `italic`, links (`a href`), `li`, and so on), and some tags should be standalone (`img`, `br`, `wbr`, etc).

ICU MessageFormat placeholders (arguments) all behave as `standalone` type placeholders, so we can make that the default type.

So the new MessageFormat might represent these concepts like this:

```
{"link" :open html ...}Login{"link" :close html} before {$expDate}.
```

The two `link` tags are `open` and `close` type placeholders, while the `expDate` is a `standalone` type.

Note that the open/close type is not tied to the function (`html` in this case). It is a property of all placeholders, and function-independent.

So one can use it to generate a real button widget, not HTML tags.

A localisation tool would render it as:

In a section below, we will present a way to explicitly support HTML.

But the usefulness of the open-close concept goes beyond HTML, and XLIFF uses it for anything that implies structure or formatting (Markdown, Word documents, InDesign, etc.)

Using and extending

One of the main differences between the current proposal and the existing placeholders (arguments) of ICU `MessageFormat` is the ability to register custom functions.

All functions are equal

In the new standard that we are designing, MessageFormat proper will know nothing about concrete date or number formatters.
They are formatter functions, implemented like any custom function, and they have access to the same underlying mechanisms as future functions. They “just happen” to be registered from day one.

The functions can remain private (application only), be shared company-wide, and/or be shared as open source.
And they can be added upstream to the standard registry, if it is agreed to be useful enough.

A function that is added to the standard registry 10 years from now will be exactly as powerful as a legacy date formatter that entered in the registry on day one.

In Annex 1: Examples of placeholder functions we included a relatively big collection of possible extensions.

Macros

A handy addition to each message is to have “pieces” that can be reused, but are still part of the message. They “travel” together, are translated together, and deleted all at once.

They can be attached to a simple message, or to a selector message.

Unlike the translation meta (see below), this information is needed at runtime, so it must be stored in the message itself.

We will use the `{>...}` syntax to declare such a reusable piece, which we will call a “macro”, and we will use the syntax `{#...}` to invoke it. We don’t allow invoking macros in the definition of another macro.

Some use cases below.

Reuse for consistency and reduced verbosity

There is no need to repeat the same description of the argument styles and keep it in sync across message variants.

In this example, the date and amount will be rendered the same for all plural variants:

```
{>amount {$toPay currency trailingZeroDisplay:HIDE_IF_WHOLE}
{>date   {$expDate date skeleton:yMMMd}
{ [ {$dayCount plural} ]
  [   =1] {You are due {#amount} on {#date}, which is tomorrow.}
  [other] {You are due {#amount} on {#date}, {$dayCount} days from now.}
}
```

The same functionality without macros:

```
{ [ {$dayCount plural} ] ,
  [   =1] {You are due {$toPay currency trailingZeroDisplay:HIDE_IF_WHOLE}
on {$expDate date skeleton:yMMMd}, which is tomorrow.}
  [other] {You are due {$toPay currency trailingZeroDisplay:HIDE_IF_WHOLE}
on {$expDate date skeleton:yMMMd}, {$dayCount} days from now.}
}
```

Note how, without macros, a developer needs to repeat (and keep in sync) all those styles.

And the benefit of macros is even more noticeable when even more repetitions occur for languages with more than two plural forms.

Reduce the combinatorial explosion from multiple selectors

Note: this is a **bad internationalization practice**, and a frequent source of bad translations. Developers have no way to judge if the independent pieces in the source language will still be independent in all target languages.

So this use is contentious, and needs discussion with the MFWG.

For languages with many plural forms (for example Arabic, with 6), combining two plural decisions in one message results in 6 * 6 = 36 combinations. Add a gender (3 options) and the number of message variants goes to 6 * 6 * 3 = 108.

But there are valid cases where such constructs are useful. As long as the pieces are independent, they will not cause unintended consequences.

For example: *"This hotel has 200 rooms, 30 suites, accommodating up to 600 guests"*.

The various features are independent, so the message would work relatively well to build from pieces.

```
{>rooms  {[{$roomCnt plural}]  [=1] {{$roomCnt} room}   [other] {{$roomCnt} rooms}}
{>suites {[{$suiteCnt plural}] [=1] {{$suiteCnt} suite} [other] {{$suiteCnt} suites}}
{>guests {[{$guestCnt plural}] [=1] {{$guestCnt} guest} [other] {{$guestCnt} guests}}
This hotel has {#rooms}, {#suites}, accommodating up to {#guests}.
```

Yes, it introduces through a "backdoor" the current (bad i18n practice) situation in which developers build complex messages through what is effectively concatenation.

But here are some benefits:

1. We already agreed that some kind of message reference is needed, see issue #127

2. The developers can always do this "behind our back" by defining and formatting separate messages, then using them in the final one.
However, by creating this in a standard way, we make it visible to linting, refactoring, etc.

3. All the pieces needed to build a message travel together for localization.

This means better context for translators. They can see all the pieces that go in the final message.

Localization tools can preview the resulting messages as one, validate them, spell check, etc.

Better isolation (the same part does not get reused)

One of the early decisions of the workgroup was to disallow internal selectors, and only allow message level selectors.

For example, we want to only allow the selector `roomCnt` to allow message-level selection:

```
{ [ {$roomCnt plural} ]
       =1 {Our hotel has {$roomCnt} room, cleaned daily ...}
    other {Our hotel has {$roomCnt} rooms, cleaned daily ...}
}
```

and we should disallow this – selection occurring over a concatenated sub-message:

```
Our hotel has
{$roomCnt plural =1 {{$roomCnt} room} other {{$roomCnt} rooms}},
cleaned daily ...
```

In our meetings, we discussed that users may insist on concatenated selector sub-messages. And regardless, users will still need to be able to migrate their old ICU MessageFormat messages written in this style to the new standard. And the way to address it was through message references (Oct 19, 2020 meeting notes)

The direct implementation of that idea (using message references) would look like this:

```
roomMsg = {$roomCnt plural =1 {{$roomCnt} room} other {{$roomCnt} rooms}}
mainMsg = Our hotel has {&roomMsg} cleaned daily ...
```

In the above two *separate* messages, `mainMsg` directly refers to the `roomMsg` message (what we named “message reference”). The `&` in the syntax is not part of the current proposal, it is for illustration.

But constructing messages like `mainMsg` that use message references will result in translations with grammatical errors. The following explains an example:

> Sometimes text that is “outside” a selector needs to move “inside” for a correct translation.
>
> In English the result would be *“...1 room cleaned daily…”* and *“...20 rooms cleaned daily...”*.
>
> In Romanian, *“cleaned”* would need to be *“curățată”* (singular) or *“curățate”* (plural).
>
> To make it linguistically correct a translator would bring *“cleaned”* inside the selector:
>
> ```
> roomMsg = { [ {$roomCnt plural} ]
>   [   =1] {{$roomCnt} cameră curățată}
>   [ many] {{$roomCnt} de camere curățate}
>   [other] {{$roomCnt} camere curățate}
> }
> mainMsg = Hotelul nostru are {&roomMsg} zilnic ...
> ```

> The result looks good in context, but now our translation memory (TM) contains a translation from *"# rooms"* to "*# rooms cleaned*" in Romanian.
> And when that TM entry is reused in other messages, also without context, it results in incorrect translations.

We solve this problem by defining these "sub-messages" as macros inside the main message, not as standalone messages accessed with references.

In this way, they are not reusable from other messages. They can only be used from the message in which they are defined. They "travel" with the message using them, and they get translated / removed with the message. So they are always used in the same context.

We can think of the macros as local variables instead of global variables (full fledged messages).

All the message parts are accessible together

In the example in the previous section about message references, if `mainMsg` changes in the source language, while the message it refers to (`roomMsg`) doesn't change, then typically only `mainMsg` is sent out for translation as a "delta" (the only message that changed).

But the quality is better if we can send out all the components, as a change in the main message may require changes in the parts that it refers to.

These two approaches are conflicting, with price and time restrictions often dragging down quality.

Macros offer a compromise, where all the components of a message are always available together.

Some syntax ideas

To reduce clutter, we can choose to only give the macro a new name if we need to use the input more than once with different formatting options (otherwise it would cause ambiguity by "masking" the original parameter).

Example:

```
{>drivingTime duration hour:short min:short}
{>distMetric   {$distance measurement system:metric}}
{>distImperial {$distance measurement system:imperial}}
The San Francisco - LA distance is {#distMetric} ({#distImp}), and would take
about {#drivingTime}.
```

The result might look something like this:
"The San Francisco - LA distance is 616 Km (383 miles), and would take about 5 h 30 min."

We defined `distMetric` and `distImperial` with different names based on `distance` with different styles. We also defined `drivingTime` as being `drivingTime` input plus a style, but we didn't have to give it a different name.

Therefore, this definition:

```
{>drivingTime duration hour:short minutes:short}
```

is just a simplified version of this complete definition syntax:

```
{>drivingTime {$drivingTime duration hour:short minutes:short}}
```

The Registry: syntax exploration

Draft example of a registry entry for a formatting function.

This is not a complete example, only a sketch to show the direction and main ideas.

```
{
  "name": "datetime"
  "description": "This function formats the argument as a date or time"
  "input": "Any type that can represent a moment in time:
      long (epoch millisecond), java.util.Date, java.sql.Date,
      java.time.Instant, ISO 8601 string, etc."
  "options": {
    "skeleton": "string using the ICU syntax (for example 'yMMMd')"
    "year": ["2digit","numeric"]
    "month": ["2digit","numeric","short","long"]
    "day": ["2digit","numeric"]
    ...
  }
},
{
  "name": "nameformat"
  "case": {
    "l10n": "enum"
    "values": {
      "ro": ["nominative","genitive","dative","accusative","vocative"],
      "ru": ["nominative","genitive","dative","accusative",
             "instrumental","prepositional"],
      ...
     }
  }
  "politeness": {
```

```
    "l10n": "enum"
    "values": ["casual","formal"]
  }
  "gender": ...
},
{
  "name": "html"
  "onclick": {
    "l10n": "hidden"
  }
  "alt": {
    "l10n": "freeform"
  }
  "title": {
    "l10n": "freeform"
  }
  "id": {
    "l10n": "read_only"
   }
}
```

Using JSON syntax for this example, but the final syntax needs to be determined: probably either JSON or XML.

The exact “schema” of the registry is TBD.

The registry should be machine readable.

But it is only intended to better integrate between libraries, programming languages, and with localization tools.

The full descriptions of the functions can live in stand-alone documents.
This is similar to MessageFormat in ICU today, for which the full description of the flags, skeletons, and arguments for number or date formatters are documented in the ICU User Guide rather than in MessageFormat proper.

The kind of information we would store in the registry:

- Formatting functions definitions (including input type, parameters)
- Selection functions (including input types, parameters, matching keywords if not “free form”)
- For each parameter, list the possible values (if enum-type)
- Default functions based on argument type
  This is for convenience, when using placeholders without function name: `"Your bill`

`is due on {$dueDate}"` ⇒ use a date formatter if the parameter is `Date`.
For example numbers (`int`, `long`, `double`, `float`, …) => `number` formatter function, `java.util.Date` => `datetime`, and so on.

- Default options for each function, if not specified
(for example `{$dueDate datetime}` ⇒ `style:LONG`)
- Translation metadata keys & values (see below)

Options must also contain information for localization tools / translators (not captured here).

Some localization attributes for options:

- **Hidden** (the attribute is not visible to translators; that is, it is useful for runtime, but not useful for localization)
- **Read only** (the attribute can be seen by translators, but not changed)
- **Enum values** that can be selected by translators, but not free-form translation
(for example grammatical cases)
- **Free form** (?). Probably only for things that represent good standalone text, not fragments.
For example an `html` function might have `alt` and `title` attributes that are localizable.

Some examples:

Hidden: `Login {"a" :open html href:"..." onclick:"..." id:"...">here{"a" :close html}!`

The open `<a>` tag would result in an open placeholder, and it is useful to know it is a link. But the `onclick` handler is not relevant for translation, only adds noise, and damages the leveraging.

So for the custom function `html`, the `oneclick` attribute is declared **hidden** in the registry.

Read-only: the translator can see the attribute, but can't change it.

The `id` in the example above would be read-only (one might argue that it can be hidden).
The url (`href`) can be read-only or not (see below).

Enum values: `Hello {$user nameformat length:short usage:addressing style:informal}!`

The translator might change the `style`, `usage`, or `length`, but they can only choose from a predefined set of values (`formal` / `informal`, `long` / `medium` / `short`).

Note: attributes inspired from the draft "CLDR Person Name Formatting" document.

Free form: already mentioned the `alt` and `title` attributes in html.

`See <img src="images/dlg_cfg.png" title="Screenshot of the configuration dialog">.`

In some cases even the URLs (`href`) might also need to be “translated”, when the URL points to something to a site that can’t be determined algorithmically. For example the Russian Wikipedia page for Bruce Willis is https://ru.wikipedia.org/wiki/Уиллис,_Брюс.

These kinds of attributes might also be represented as macros, which takes them out of the main “text flow.” So they are not attributes anymore.

The choice of which options are presented for translation may depend on the locale, but the set of possible valid options is specified in the registry. That set, however, may be expanded over time.

In CLDR, BCP 47 also provides for aliases, deprecation, and version dates, which allow for more control over the identifiers over time. We should also have a mechanism for these features.

An attribute can also contain examples, which may also be locale dependent (for example the month short can be “Jan” in English and “янв.” in Russian). The examples might be added by developers or by tools (for plural examples, maybe inflected names, other).

Discussion of the registry has so far been deferred in the current MessageFormat v2 effort, but there is agreement that it should exist, and that it is the way to extend the functionality in the future.

There can be company or product specific registries that can be sent to localization, together with the source content to be localized. They can contain entries for custom functions.

But there is only one standard, official registry that contains functions that all libraries that claim to be compliant should implement. It would initially contain the current functions (datetime, number, duration, and plural, selectordinal, select, etc).

The official registry is expected to grow in time, as we add more functionality (for example, list formatting, intervals, people names formatting (deal with formal / informal preferences), etc.

Even message references can be implemented as functions and added later
(see Annex 1: Examples of placeholder functions – Message references).

Translation Metadata

Both messages and placeholders will have additional info.

Some of that info is needed at runtime, and some is only useful during translation and/or development.

Translation Metadata is information that is only needed for translation and can be dropped for runtime.

This was not added in the data model presented above to keep the level of detail in this document appropriate for its purposes.

It is probably enough to be a map, so this is the proposed data model:

```
L10nMetadata ::= map<String, ?>
```

The registry would contain descriptions for the elements (keys and values).

TBD if the values are flat, or can be richer structures.

Metadata can be “attached” to the message. Internal to the metadata, there can be information for placeholders.

Some examples of localization info to store in meta:

- Comments for translators
- Links to screenshots
- Validation restrictions
    - Length (for example visual width in “em”, or storage width (40 bytes in UTF-8, or 40 code units in UTF-16)
    - If it is acceptable to delete a placeholder or not
      For example if it is OK to replace “`Hello {$user}!`” with “`Hello!`”
- Examples for the final message.
  For example the `user` placeholder in “`Hello {$user}!`” might have `l10n_meta:{example:["John", "Mary"]}`.
- Disambiguation and context data (for example *“Print”* as a button vs a title)
- For some more examples see Internationalization Tag Set (ITS) Version 2.0
- Placeholder flags: `canCopy`, `canDelete` (see the XLIFF 2.1 spec for use cases)

All this information is needed for translation only, and is not useful at runtime.

There is nothing equivalent to the concept of translation metadata in the ICU `MessageFormat` today. So some might find it tempting to leave it out of the specification.

But most systems would benefit from such additional data, and many existing systems have some improvised, non-standard ways to do it.

Therefore it would be beneficial to have it in the specification, and in the registry, in order to support interoperability between implementations, refactoring, exporting to localization tools, etc.

Part of our group's agreed-upon deliverables is a standard mapping to XLIFF.
We should clearly say that messages carry this kind of additional information, and what is the standard way to represent it in XLIFF, regardless of the syntax in the sources.

That encourages localization tools to expect it and use it (show previews, link to images, validate using that information, etc).

It is currently unclear how to represent these metadata in the syntax itself without cluttering it. It might be best to have it in comments, which makes it dependent on the message storage?

For example Java supports storing messages in: .property files, .xml files, or classes (`java.util.ListResourceBundle`)
So one can imagine storing the meta in comments:

.properties

```
/*
 * Temporary, check with UX
 *
 * @l10n.desc: Message showing only once, at start time
 * @l10n.screen: http://foo.com/screenshots/fooProject/helloDialog.jpg
 * @l10n.param user {example: John, canDelete: true}
 */
msg_hello = Hello {$user nameformat}!
```

vs

.xml

```
<!--
 * Temporary, check with UX
 * @l10n {
 *   @desc: Message showing only once, at start time
 *   @screen: http://foo.com/screenshots/fooProject/helloDialog.jpg
 *   @param user {example: John}, canDelete: true
 * }
-->
<entry key="msg_hello">Hello {$user nameformat}!</entry>
```

Or even using a separate namespace in XML with more structure, and using ITS 2:

.xml

```
<mf2_l10n:entry desc="Message showing only once, at start time">
  <screen url="">http://acme.com/screen/projectId/helloDialog.jpg</screen>
  <param id="user" example="John" canDelete="true"/>
</mf2_l10n:entry>
<!-- Temporary, check with UX -->
<entry key="msg_hello">Hello {$user nameformat}!</entry>
```

Each of these might feel more natural for various tech stacks (a JavaScript library would probably prefer something more like JSON).

These are easier to describe in a data model, but forcing a fixed syntax will result in an “alien” look for some users.

They are also easy to convert between formats. For example, one can choose to load Android `strings.xml` files and export them as iOS `.stringsdict` files.

And the existing resource compilers already discard all this info. This means that there are no changes necessary, increasing the chance of adoption.

These localization directives, when they are somewhat structured, have proven useful with most previous formats that use them.

TBD if we want to decide on an exact format, or just say it is a map, with the keys and meaning described in the Registry, and leave it to framework-specific implementations to decide the exact syntax. As long as one can algorithmically load / save them, it would not matter much.

Design Considerations

One single message vs. “bundles” of messages

One of the design considerations is whether a message is a self-contained entity or a “bundle” of messages that can include groups and other messages.

The latter is more powerful, but a number of practical considerations have led to the design proposed in this document.

That is, a message is normally self-contained, but for those implementations that want to support a collection of messages, external functions can be used to provide that ability.

An independent specification might add such higher level structures, if needed.

Here are some of those practical considerations...

Mixed technologies

Bigger applications are usually put together from many components. Some merge libraries (or shared libraries) into the main application, but in some cases, the pieces come together as a mixture of client application + server side.

A server side application might use a mixture of C++ and Java, with some of them controlled by the main app developer, and some being libraries that come from repositories like Maven, with clients written in Swift, Kotlin, JavaScript, or something else, from repositories like npm.

It is likely for all these components to use different localization technologies from each other.

For example a native Windows / MacOS application might use native OS techniques for localization, but use a library that uses `gettext`, and string provided from a server side database.

So the idea that one message can reference another, and that the translator or a lint tool has access to all messages, does not work for these kinds of applications.

Adoption

Attempting to force a developer of an application to change localization technologies would severely impede adoption.

For example, let's say that an application uses the Android provided localization stack. Which uses .xml files, standard methods to load messages, and so on, a complete ecosystem.

Now, let's say that there is a Greek grammatical problem that the new Message Format standard can help solve. Now, the developer would need to move their strings from the Android way of doing things to a new tech stack: another file format for string storage, update all the code the new loading APIs, not just the formatting, and finally change their localization process to translate the new file formats.

And in the process, the developer would give up all benefits that the underlying platform offers (locale negotiation and fallback, direct use in layout files without explicitly code calling, language-splits that are downloaded on demand, etc).

It should be possible to just store a string in the existing file format, load it with the existing APIs, and use that string with the new formatting APIs to fix the bug. Without a full migration.

Message IDs not available at runtime

Also, message references can be hard / impossible when using native formats.

Windows and Android use numeric IDs for strings (and other resources).

On Windows this looks something like this:

```
#define IDS_HELLO 100
#define IDS_CITY 100
STRINGTABLE
BEGIN
    IDS_HELLO "Have you visited %s?"
    IDS_CITY "San Francisco"
END
```

If one would want the `IDS_HELLO` message to refer to `IDS_CITY` at runtime there is no good way to do that without majorly changing how resources are loaded and handled.

There is no way to make a reference (`"Have you visited {&IDS_CITY}?"`) because `IDS_CITY` does not exist anymore at runtime as a string identifier `IDS_CITY`.

The developer-friendly identifiers of the messages are also dropped on Android, and even for JavaScript, when minified or obfuscated.

And there are platforms that don't use identifiers at all for localization.

Take for example Angular: `<h1 i18n>Hello world!</h1>`

Or gettext: `gettext("Hello, world!")`

Although in some cases, the English string is considered to be the identifier, it can't really be used as a reference for a message. It is easy to break, and using the string itself is like using the message, not a reference to it.

One can argue these are bad practices.

But many applications are written this way, and trying to move them to a system that requires identifiers means there will be friction, and this might reduce or prevent adoption.

Changing the resource format is an all or nothing proposition, which requires global code changes, changes in the localization system, and migrating translation memories done all together. It cannot be done string by string, as one can do just to fix bugs.
And similar to what was mentioned before, changing the resources also breaks all the functionality one gets "for free" from the existing platforms, things like locale negotiation and fallback, download languages "on demand", etc.

Of course, nobody can predict if the friction will completely prevent adoption, or just slow it down. But the more friction we have, the less likely it is that what we design is adopted.

Also, big projects don't have all the strings always accessible to translators. They often just export only the new or changed strings. So translators don't have access to modify another string to make it work well with a new one. Worse, they might not even want to do that, as sensitive content (ex: legal, medical), once it gets approved, can't easily be changed.

Message groups and references should be optional, but not required for the functionality.

The need for references to other messages is reduced (eliminated?) by the introduction of the macros, as explained in the Macros section.

It can also be added later on with registered functions (see Annex 1: Examples of placeholder functions – Message references).

Annexes

Annex 1: Examples of placeholder functions

This annex is intended to show how flexible formatting functions can be, to inspire confidence in the extensibility of the proposed model.

There are no “privileged” functions.

A function that is added to the standard registry 10 years from now will be exactly as powerful as a date formatter “grandfathered” in the registry on day one.
They have access to the same data, have the same signature, and are invoked the same way.

Not one single function is part of the standard, they all live in the registry. But there is a standard registry and custom / proprietary registries.

It is like a plugin architecture, where first party plugins don’t have any advantage over third party plugins.

This annex is split into two sections, “Probably desired functionality” and “Showing the power and flexibility.”

The split in the two sections is somewhat opinionated. But it was done only to include a bit of structure.

Since all functions are equally powerful, and implemented using the same mechanism, it does not matter in what section they are included now.

Probably desired functionality

Protected text

Used to protect a message fragment from translation, while still keeping it in the main text flow, and having it rendered at runtime.

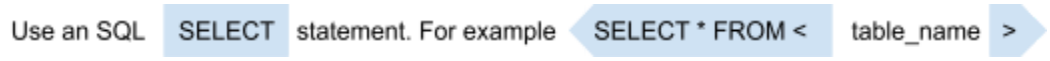
For example if we want to translate this message:
“Use an SQL `SELECT` statement. For example `SELECT * FROM <table_name>`”

And want to be sure that the localization does not mistakenly translate the SQL code.
We can protect it like this:

```
Use an SQL {"SELECT"} statement. For example {"SELECT * FROM <"
:open}table_name{">" :close}
```

We can of course pass the parts that we want protected as parameters, but this is simpler, and the translator gets to see the full message.

A localization tool would render this text as:



The text in light-blue rectangles can't be changed by translators, and the `:open` / `:close` tell the l10n tools to forbid changing the order of `SELECT * FROM <` and `>`.

This is a feature already available in many localization tools.

References to other parts of the message

When building messages we are often faced with the challenge of grammatical agreement, where part of a sentence must change (inflect) depending on attributes of other part(s) of the sentence.

For example in Romanian "*blue*" would be translated as "*albastră*" (feminine singular), "*albastru*" (masculine singular), "*albastre*" (feminine plural), "*albaștri*" (masculine plural).

So in a sentence like this: "`The {$item} is {$color}`" the color would need to agree in gender, number, and case with the item.

This needs to be worked out.

Direct (const) values

The "thing to format" can be a constant rather than a parameter, something we know when we write the code and at build time.

Localization is (usually) done in one, maybe 2-3 language "flavors", but some languages are used in tens of countries, with different conventions. Think Arabic, English, French, Spanish.

These elements have region dependent rendering, or might even depend on user settings.
So the translators cannot "translate" these elements, even when known.

Some examples: number / date formats, hour cycle, calendar, etc.

This can be a convenience, where the "name" of the placeholder is a language-independent form that the formatting function can parse:

```
A free {"30" number} days trial.
Next meeting at {"20211231T143000" datetime skeleton:yMMMMdjm}
```

Today developers have to treat constants as if they are unknown values, and pass them as parameters for the formatters.

Formatting tags

Years ago software strings were plain text, with no formatting.
There was a clear delimitation between software strings and documents (with formatted text).

But modern technologies allow for richer user interfaces. Software strings need to be able to encode style information: links, bold / italic, color, font sizes, headings, alignment, etc.

So it is important to have a way to represent these concepts in a way that is not an afterthought.

At the same time we don't want to tie MessageFormat to HTML and nothing else.

For this we are taking inspiration from XLIFF, which has a long experience as an interchange format that can handle formats like HTML, InDesign, MSWord.

What is needed is the concept of `open` / `close` / `standalone` placeholders (already covered).

Many translation tools are already able to handle these kinds of placeholders, preventing incorrect reordering or nesting.

For example we can register a function (`html`) that creates HTML tags:

```
Click {"link" :open html tag:'a' href:'....'}Login{"link" :close html} to
create an account.
```

Macros (see the Macros section) can also help to minimize the noise inside the message, and allow the flexibility to change the html code without changing the message:

```
{>link html tag:'a' href:'...' style='...'}
Click {#link :open}Login{#link :close} to create an account.
```

And the `html` function can generate `<a href:'...' style='...'>` if the placeholder type is `open`, and `</a>` if the type is `close`.

It can also be intelligent enough to generate no tags if we want to generate plain text.

Or we register a markdown function under the `html` name, and that would generate markdown instead of html.

Or something like Java AWT can create a button for *"Login"*, with everything wrapped in a `FlowLayout`.

We leave a lot of flexibility for the implementation and extension, and at the same time we have all the information needed for localisation and validation.

Showing the power and flexibility

One valid worry is that we might design the placeholders in a way that is too rigid, locking ourselves in a situation where in a few years we might need some features that we can't add in a backward compatible way.

So below we will show some examples of functionality that today we might consider not-recommended practice, but can be added within the current proposal, with developer functions.

References to other messages

```
msgId = Click {$btnOk msgRef} to continue
// Usage
msgText = resources.load(msgId)
msgText.format( { "btnOK" IDB_OK } );
```

In this case, the developer can store the message ID in the `btnOk` parameter, which can be integer, string, enum, or something else. And the `msgRef` function, which is implemented in a platform dependent way, will use platform native APIs to load the referred message.

But it is a function, added to the registry, and not an integral part of the standard.

If the underlying platform supports string identifiers for messages we can combine this with the "const values" idea, and there is no need to pass the identifier as a parameter:

```
msgId = Click {"btnOk" msgRef} to continue
// Usage
msgText = resources.load(msgId)
msgText.format();
```

Combined functions

Not saying that it is a good idea, but the flexibility is there:

```
"... {$argument combine
  func1:'convert' funct1.from:metric to:imperial
  func2:'measureformat' func2.style:long
}"
```

The function `combine` would retrieve (based on name) the `convert` and `measureformat` functions, and invoke them each with the options from `funct1.*` and `func2.*` respectively:

```
function combine (argument, options) // pseudocode
```

```
var function1 = registry.getFunction(options[func1])
var tmp = function1.invoke(argument, funct1 options]
var function2 = registry.getFunction(options[func2])
return function1.invoke(tmp, options[func2 options]
```

This is just to show that we still have a lot of flexibility to do advanced things even with such a simple model.

Annex 2: Multi-selector matching algorithm alternatives

The Multi-selector matching algorithms section "glanced over" the exact algorithm use to determine the best match for tuples of conditions + keys.

Here we present a few options.

Option 1: behave the same as ICU

We filter all options that match the first selector, then filter out all options that match the second, and so on.

But this is less than ideal, as we can see in this example:

```
{ [ {$itemCount plural} {$itemGender gender} ]
  [     =2                 other ] { ... the message 1 M ...}
  [    one                 other ] { ... the message 1 M ...}
  [  other                 other ] { ... the message O O ...}
  [     =1                   fem ] { ... the message 1 F ...}
  [    one                  masc ] { ... the message 1 M ...}
}
```

With an input of `{itemCount: 1, itemGender: male}` the first selection (`plural`) will match best with the `[ =1 fem ]` message. Then the second selection (`gender`) will have no match, resulting in an error.

This would be the ICU behavior today, with nested selections, but with an error only if ICU didn't require an "other" option at each level. With the required "other", ICU is still forced to sequentially select a possibly sub-optimal combination.

It is pretty clear that the `[one masc]` entry is a much better match for our input.

Option 2: filter and sort

This is an algorithm that gives better results while still being intuitive and non-surprising for a non-technical external observer (think translator, etc).

We take each column, and we remove the entries that don't match. Keep the "other" catch-all entry. We need to define how catch-all entries work for other types of selection, such as for gender.
Then we sort the remaining entries, with keys at the left having higher priority than the ones to the right.

In the example above, with `{itemCount:1, itemGender:male}` input, we filter by column, `itemCount: 1`:

```
   one              other { ... the message 1 M ...}
 other              other { ... the message O O ...}
   =1                 fem { ... the message 1 F ...}
   one               masc { ... the message 1 M ...}
```

Now we filter by the second column, `itemGender: male`:

```
   one              other { ... the message 1 M ...}
 other              other { ... the message O O ...}
   one               masc { ... the message 1 M ...}
```

And we sort, with leftmost keys having priority (think lexicographic sort):

```
   one               masc { ... the message 1 M ...}
   one              other { ... the message 1 M ...}
 other              other { ... the message O O ...}
```

The top entry `[one, masc]` is our best match.

Using some kind of scoring might also work, so that there is no need for real filtering or real sorting, and would improve performance.

But that is an implementation detail.

Option 3: calculate a distance

For example, for plural selection, take advantage of the fact that =1 is better than "one", when matching the input value 1. There is example code implementing such an algorithm.

Not complicated for someone technical, but not simple to explain in a few paragraphs.

But the results are not surprising. Which is what we want when we look at the results not as something to understand logically, but something you tend to just ignore when it all works "as expected" (the "principle of least surprise")

But of course we need to check if others also find the results to be as expected.

Annex 3: Alternate syntaxes for multi-selectors

The syntax presented in the Multi-selector syntax section is derived directly from the current ICU syntax, replacing the selectors and “keys” with arrays of selectors / keys (represented by square brackets).

It is useful to explain the transition from single to multi-selection, but it is very verbose.

But it can be simplified without introducing any ambiguity in parsing.

Below we explore some ideas.

Option 1:

Eliminate the square brackets:

```
{ {$count plural} {$host_gender gender}
     =1 female {One guest to her party}
     =1   male {One guest to his party}
     =1  other {One guest to their party}
    other female {{$count} guests to her party}
    other   male {{$count} guests to his party}
    other  other {{$count} guests to their party}
}
```

It can still be parsed unambiguously, keeps style consistency with ICU, and “degrades” to a pretty clean form when for one single selector:

```
{ {$count plural}
     =1 {One guest}
    other {{$count} guests}
}
```

Option 1a:

Similar to the one above, but replace the `{}` wrapping the selectors with `()`.

```
{ ($count plural) ($host_gender gender)
    ... Same as Option 1 ...
}
```

and reduced to one selector:

```
{ ($count plural)
```

```
    =1 {One guest}
   other {{$count} guests}
}
```

It is not that consistent with ICU, but is more readable (too many nested `{}` can be hard to parse by the human eye).

Annex 4: Alternative for selection depending on a formatter result

“Declare” the placeholder in a common place, and return info for the selector function

```
{>amount number trailingZeroDisplay:HIDE_IF_WHOLE}
{ [ {#amount plural precision:4} ]
       =1 {{#amount} dollar}
    other {{#amount} dollars}
}
```

The `{>...}` syntax declares a “macro” that is then used everywhere (with the `{#...}` syntax).

At runtime the message “interpreter” would invoke the number formatting function, which would return 2 pieces of information: the formatted result, as string, and some extra information to be used by the selector function.

The main drawback of this approach is that we need to specify what “pieces of information” are needed by the selector, and how to represent that info.

Then all formatters that can be used for plural selection need to return that info.

Also, formatters might also need to return information needed for other selectors, not just plural.

But because the formatter function does not know what selector will be used with it, it would need to return info for all possible selectors: precision, gender (think a list format).

The prime example is plural selection, where the set of parameters to be communicated between formatting & selection has gone up a few times as we learned more about how this works across languages and with non-integer numbers.

Annex 5: Expanding selection cases

There is one main friction point today between localization tools and plural functionality in ICU `MessageFormat`, plurals in `gettext`, Android (`<plurals>`), `NSLocalizedString` on macos, etc.

The problem is that the localization tools are designed to translate 1:1 relations: one source string maps to one target (translated) string.

Some systems resolve this by processing the messages before passing them to the I10n systems, and expanding all the selectors that would result in an n:m mapping.
For example plurals, where a message with `one` / `other` message variants in English would need to add options for `few` and `many` if the target language is Russian.

This is too much to cover in this document.

But the short answer to the problem is that we will work on this with the XLIFF standard and the localization community, and our mapping from MessageFormat 2 to / from XLIFF will “just work”.

We already have a proposal in the works for an XLIFF module, already reviewed by a few people with XLIFF and L10N tooling expertise:

- Plural and gender support in XLIFF 2.x
- XLIFF 2.x module to support plural / gender / select (draft)

Annex 6: Complications

Dependencies

Selection has a tricky feature, whereby the selection depends on the formatted result. Example: the argument value for `$length` is 0.92 meters. We format that for the US for a particular usage, and it is converted into 1,006124 yards. With the selected number format settings, it will format as “1.0 yards”. That has plural-category = other, and gender = neuter.

1. Note that if the number format settings had no decimals, it would format as *“1 yard”*, plural-category=one. So if we are selecting on the basis of the `$length` argument to get the right submessage, *we have to first format the value **before** doing the selection in order to get the correct plural category for selection*. (This is the case even if the resolved unit is *“1 meter”* vs *“1.0 meter”*).

2. Pretend that English had gender like German. In that case *foot* is masculine, *yard* is neuter, and *mile* is feminine. Based on the magnitude of the input `$length`, we could get any of these three in the formatted unit. *So if we need to select on the gender of a unit value to get the right submessage, we have to format **before** doing that selection*.

3. But what if 2 submessages have different formats for a given argument? What ICU does now is ignore all but the first one for the purpose of selection. This may be unexpected behavior. Moreover, a single submessage can have the same argument used in two different placeholders, as in the above example `"At {1,time,::jmm} on {1,date,::dMMMM},...`. So the selection process has to know which format to use. And in a strange case, might need to select on the same argument, but with 2 different formats.

Robustness

Because plural categories are a fixed set with a defined fallback (`other`), if the plural categories for a language change, then the result is still well defined in MF1.0. In the localization process, a selection message for plurals with 2 submessages in English will be expanded to up to 6 different submessages (for Arabic) or reduced to 1 submessage (for Japanese). The order of the submessages is not significant, and the translator has no ability to change the order of the messages in any event.

IMO, it would be dangerous to let them change the order if it were significant, since they are unlikely to fully understand all the ramifications of ordering; moreover the ordering might differ from message to message.

It is important that the messages retain a certain stability over time. For example, if the plural category `many` is added to the French rules (it happened), then a formerly-valid French selection message still works reasonably well, even though it doesn't have a submessage for `many`. Any value that is not mentioned specifically will fall back to `other`. Even if the message is not optimal, the situation is no worse than before. Over time, if desired, the `many` case can be added to the product's messages.

This situation does not happen with gender MF1.0, because they are simply strings. The genders can't be expanded to 4 for the case of Polish, or reduced to 1 for Hungarian. And if the genders for a given language were expanded, there would just be a miss (check this; maybe ICU uses the last submessage?) or have an order dependency.