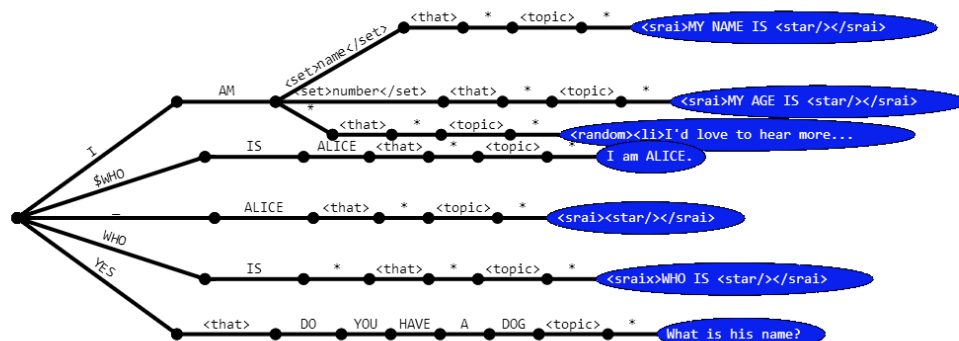


## AIML 2.0 Working Draft

## AIML 2.0 Working Draft



Revision 1.0.2.22

March 9, 2014

Richard S. Wallace

ALICE A.I. Foundation

Contact: info@alicebot.org

**1. Introduction**

This document is a draft specification for a new AIML (Artificial Intelligence Markup Language) standard, version 2.0 of the language. AIML is an XML language for specifying the contents of a chat robot character.

An AIML Interpreter is a program capable of loading and running the bot, and providing the bot's responses in a chat session with a human user, called the client. This document explains in detail both the syntax and semantics of AIML, as well as key features that should be supported by an AIML interpreter.

The primary design goal of the original AIML language was simplicity. AIML is motivated by two observations:

1. Creating an original, believable chatbot character requires writing a significant amount of content, in the form of conversational replies(\*)

(\*) - This proposition may not be true for chatbots based on other technologies. AIML implements a form of supervised learning, where a person, the botmaster, plays a crucial role in training the bot. Unsupervised learning systems, on the other hand, attempt to teach a bot through conversations, in effect crowdsourcing the bot content. The unsupervised model has its own drawbacks however. Specifically, the bot database becomes filled with nonsense, which then an editor must later delete. The tradeoff between supervised and unsupervised methods might be summarized as "Creative writing vs. deleting garbage."

2. The people who are most suited to writing the bot content are not, by in large, computer programmers. Those with literary backgrounds are more skilled at developing content for original characters(\*).

(\*) The caveat to this observation is that there are of course, some talented people who have mastered both computer programming and the literary skill to write quality chatbot content.

When AIML was first designed in the late 1990's, the World Wide Web had burst upon the stage and a rush of creative energy was poured into building websites. This tsunami of activity has in fact continued to this day. What has changed however is that the web lost its original simplicity. Perhaps it was inevitable as users demanded more and more sophisticated services through the web, that layers of complexity would be added. In 1994 however it was possible to author a web site with only rudimentary knowledge of a few HTML tags.

Because at that time, a number of creative people had mastered the then-simple HTML, I made a decision to create an equally simple AIML. I was fond of saying, "anyone who knows enough HTML to make a website, can learn enough AIML to write a chatbot."

A parallel development beginning in the 1990's was the development of XML, including specifications, standards, documents, tools, and applications for XML. Perhaps the world has not gone the way that the XML evangelists hoped in the 1990's, as its many competing formats remain viable today. But XML has not gone away either. It remains true that XML is a broadly supported standard, and its tag-based representation is easy to grasp without sophisticated knowledge of computer science. AIML authors have found the many

XML tools, such as DTDs, syntax checkers, and editors, to be useful when creating bots. For these reasons AIML 2.0 remains hitched to the XML wagon.

At some level however, AIML does not depend on XML syntax. There is a deeper representation of the data we represent in XML files. As long as the representation can capture the basic structure of a pattern path (the input pattern, that pattern and topic pattern), and a hierarchical response template, then AIML could be written in a number of different formats, including Lisp S-expressions, JSON, or a structured text format. The AIML 2.0 draft even includes an alternative representation: a hybrid of flat files and XML called AIML Intermediate Format (described in a section below).

Modifying AIML inevitably reduces some of its original simplicity. Adding more tags and more features make the language more difficult for people to understand. The urge to keep it as simple as possible is tempered by our experience over the past decade, in which AIML botmasters learned that the language had some serious limitations. AIML 2.0 is an attempt to address the shortcomings, while balancing the original goal of keeping the language as simple as possible. This AIML 2.0 draft specification is, for the most part, designed to be backwards-compatible with the AIML 1.0 and earlier standards, in that way preserving the simplicity of the original language. What's new are some new features that build on top of the original language in such a way that the concepts can be pedagogically organized so that AIML can be taught in beginner, intermediate and advanced levels.

## What's new in AIML 2.0?

- **Zero+ wildcards:** new wildcards that match 0 or more words.
- **Highest priority matching:** select certain words to have top matching priority
- **Migrating from attributes to tags:** more dynamic control of attribute values
- **AIML Sets:** match inputs with sets of words and phrases
- **AIML Maps:** map set elements to members of other sets
- **Loops:** Iterations
- **Local variables:** variables with scope limited to one category.
- **Sraix:** access external web services and other Pandorabots
- **Denormalization:** the (approximate) inverse of normalization.
- **Pandorabots extensions**
  - **date:** formatted date and time
  - **request:** access previous input request history.
  - **response:** access previous bot response history
  - **unbound predicates:** check if a predicate has been set or not
  - **learn:** learn new AIML categories
  - **learnf:** learn new AIML categories and save in a file
  - **explode:** split words and phrases into individual character
- **OOB (Out of Band) Tags:** AIML extension for mobile device control

## What's gone from AIML 1.0?

- **Gossip** - never well defined anyway
- **Javascript** - The interpreter does not have to support a scripting language (to be restored in AIML 2.1).

## 2. AIML System overview

AIML defines a relationship between three entities: a human chatter called the client, a human chat bot author called the botmaster, and the robot or bot itself. In general a botmaster can author multiple bots, and each bot can have multiple clients. A system like Pandorabots provides for multiple botmasters, multiple bots, and multiple clients. An AIML system embedded in a consumer device might have only one bot and one client. The AIML standard does not specify the number of bots, botmasters or clients (except that defining AIML means we have to talk about at least one of each). The details of handling multiple bots, botmasters and clients is left up to the implementation.

Care should be taken however to manage the state of each bot and each client session.

### A. Bot configuration and state:

**AIML Files** -- Each bot is assumed to have its own set of AIML files. This collection of AIML files uniquely defines the personality of the bot character. A bot may be a clone of another bot, or may connect to another

bot through `<sraix>` (defined below) but for the purpose of defining the AIML language, the simple assumption is that each bot has its own AIML files.

**Learnf file** -- one AIML file with special meaning is the file created by the `<learnf>` tag (defined below).

When an AIML template activates a `<learnf>` tag, the bot remembers or "learns" the new category, specially, by saving it in a file given a specific name by the interpreter (for example, `learnf.aiml`). The new categories learned with `<learnf>` are global to all clients chatting with the bot, so the `learnf` file should be part of the bot's AIML file collection.

**Bot properties** -- global values for a bot, such as `<bot name="name"/>` or `<bot name="species"/>`. A multiple bot system should take care to maintain bot properties individually and separately for each bot.

**Substitutions** -- normalizing substitutions, person substitutions, gender substitutions and sentence splitters are unique to each bot. Many bots may use copies of the same substitutions, but a multiple-bot system should ensure that each bot can have its own custom substitutions.

**Predicate defaults** - Predicate values in AIML are like local variables specific to one client. Typically one thinks of client profile information like name, age and gender predicates, but predicates can be used to store any string. AIML predicates are set with the `<set name="predicate">` tag and retrieved with the `<get name="predicate"/>` tag. Predicates are specific to an individual client, but the predicates may have default values that are defined for a specific bot. There should also be a global predicate default for any predicate whose default value is not specified for a bot.

**Sets and Maps** - AIML 2.0 includes a feature that implements sets (collections) and maps. The sets members are strings and the maps define a mapping from string to string. Unique collections of Sets and Maps may be defined for each bot.

The AIML standard does not specify where or how the properties, sets, maps, substitutions and predicates are defined. This is an implementation detail left up to the interpreter designer. The values could be entered through a user interface, saved in text files or a database, or in any other format including XML and JSON, as long as the interpreter can read them when the bot is launched.

## B. Client session and state

**Initialization** -- when a client connects to a bot, before they begin chatting, the bot must initialize a client session. The client session is assigned a unique ID so that the AIML interpreter can track the state of the conversation. This is important when a single bot is chatting with multiple clients, for example a web based bot.

**Predicate defaults** -- Initialization step also includes setting predicates to the default values specified for the bot.

**Predicate state** -- The chat session must keep track of the state of predicate values. Whenever a client activates an AIML category, potentially the `<set>` tag is some predicate values may change. The interpreter must remember the predicate values through the course of the conversation.

**Topic** - The AIML topic is a unique predicate value, because it becomes part of the pattern matching process. The topic can be set with `<set name="topic">`

**Conversation log** -- Generally an interpreter keeps a conversation log of the interactions between a bot and a client. The AIML 2.0 draft does not specify how or in what format these logs are stored.

**History** -- The AIML 2.0 draft does however specify that the bot maintain, within a chat session, a history of interactions for the purpose of evaluating the tags `<that>`, `<input>`, `<request>` and `<response>`. The size of the history (the number of elements saved or remembered) is left up to the interpreter designer.

**Learned categories** -- Categories learned with `<learnf>` are saved globally for the bot (see `Learnf` file above), but categories learned with the `<learn>` tag are specific to each client. The chat session should maintain any categories learned with `<learn>`.

## C. Counting interactions and sentence splitting

The basic step of AIML pattern matching is to match one input sentence against the bot's set of AIML categories. Because inputs and responses may contain more than one sentence, AIML has adopted a particular system for counting and indexing inputs and outputs.

When the bot receives a multiple-sentence input

In general one input sentence may result in 1 or more output sentences.

`<input/>` - the current input sentence

`<input index="2"/>` - the previous input sentence

`<input index="N"/>` - the Nth previous input sentence.

`<request/>` = `<request index="1"/>` - the client's last input request, consisting of one or more input sentences.

`<request index="2"/>` - the client's 2nd to last input request.

`<request index="N"/>` - the client's Nth to last input request.

`<response/>` = `<response index="1"/>` - the bot's last response, consisting of one or more sentences.  
`<response index="2"/>` - the bot's second to last response.  
`<response index="N"/>` - the bot's Nth to last response.  
`<that/>` = `<that index="1,1"/>` - the last sentence the bot uttered.  
`<that index="1,2"/>` - the 2nd to last sentence in `<response index="1"/>`, provided it exists.  
`<that index="2,1"/>` - The last sentence of `<response index="2"/>`.

Human: Hello

Robot: Hi nice to see you!

Human: How are you? My name is Jeff.

Robot: I'm very well. How are you doing? What's up, Jeff?

Human: I'm talking to a robot

Robot: Would you like to say more about that?

Human: Sure

At this point, the bot finds a category with a response to the input "Sure". The following table summarizes the current state of input/that and request/response history at the time when that category's template is evaluated,

Entity	Normalized Sentence	input/that	request/response
Human	Hello	<code>&lt;input index="5"/&gt;</code>	<code>&lt;request index="3"/&gt;</code>
Robot	Hi nice to see you	<code>&lt;that index="3,1"/&gt;</code>	<code>&lt;response index="3"/&gt;</code>
Human	How are you	<code>&lt;input index="4"/&gt;</code>	<code>&lt;request index="2"/&gt;</code>
	My name is Jeff	<code>&lt;input index="3"/&gt;</code>	
Robot	I am very well	<code>&lt;that index="2,3"/&gt;</code>	<code>&lt;response index="2"/&gt;</code>
	How are you doing	<code>&lt;that index="2,2"/&gt;</code>	
	What is up Jeff	<code>&lt;that index="2,1"/&gt;</code>	
Human	I'm talking to a robot	<code>&lt;input index="2"/&gt;</code>	<code>&lt;request/&gt;</code>
Robot	Would you like to say more about that	<code>&lt;that/&gt;</code>	<code>&lt;response/&gt;</code>
Human	Sure	<code>&lt;input/&gt;</code>	

### 3. Migrating from attributes to tags in AIML 2.0

One odd feature of XML is the distinction between tags and attributes. Consider the HTML `img` tag in an expression like

```

```

Why was this tag developed to use an attribute, rather than a subtag like:

```
<img><src>http://alicebot.org/logo.jpg</src></img> ?
```

HTML is interpreted in a static way, but an XML language can be defined to interpret tags dynamically. For XML languages like AIML, the problem with attributes is that they are not easy to rewrite dynamically.

Suppose we want the value of the `src` attribute to vary depending on another XML expression:

```
<img><src><getCompanyLogo/></src></img>
```

The problem in XML is that you can't put an XML expression inside an attribute:

```

```

is forbidden in XML syntax.

Of course, this problem is not hard to solve with a little computer programming. The XML attribute values can be rewritten by another process writing the XML. But at least for AIML and XML languages like it, we would like to specify attribute values dynamically, and allow the botmaster to write the expressions for those values in XML.

Fortunately the problem has a simple solution: don't use attributes. Any value in an attribute can just as well be represent with a subtag as in our example

`<img><src>http://alicebot.org/logo.jpg</src></img>`

AIML 2.0 modifies the definition of every AIML tag that takes an attribute so that the attribute value can be specified with a subtag having the same name. For example:

`<get name="age"/>` may be written as

`<get><name>age</name></get>`

`<condition name="job" value="manager">Hi, boss!</condition>`

may be written as

`<condition><name>job</name><value>manager</value>Hi, Boss!</condition>`

`<date format="%D %H"/>` may be written as

`<date><format>%D %H</format></date>`

Even more generally, the contents inside the attribute tags may be any template expression, as these examples show:

`<get><name><srai>PREDICATE NAME</srai></name></get>`

`<condition><name>job</name><value><get><name>profession</name></get></value>Hi, Boss!</condition>`

`<date><format><get name="localdateformat"/></format></date>`

Care should be taken to ensure that whatever these template expressions return is a valid expression for the attribute. For example in,

`<star><index><srai>GET INDEX</srai></index></star>`

The `<srai>GET INDEX</srai>` should return a valid index number  $> 0$ .

To retain backwards compatibility, either the attribute form or the subtag form may be used in AIML 2.0. In the definitions of XML tags that follow, with a couple of exceptions noted, the attribute values may also be written in the subtag form.

#### 4. AIML Syntax

*Edit: This section should be rewritten using RELAX NG notation.*

[http://en.wikipedia.org/wiki/RELAX\\_NG](http://en.wikipedia.org/wiki/RELAX_NG)

This section makes use a variant of [BNF](#) notation to describe the syntax of AIML in detail. An XML language syntax may also be specified by a DTD or XML Schema. The BNF variant here is slightly more convenient for someone writing an AIML interpreter, and also it captures one feature of AIML that goes beyond standard XML syntax, namely the AIML pattern language.

In this BNF variant:

- Literal tag names and attribute expression are written in **Consolas Bold** font.
- Expressions and clauses are written in CONSOLAS UPPERCASE.
- The following notation is used to define an expression:

(EXPRESSION) - The expression EXPRESSION is optional.

(EXPRESSION)\* - The expression EXPRESSION may be repeated 0 or more times.

(EXPRESSION)+ - The expression EXPRESSION may be repeated 1 or more times.

EXPRESSION3 ::= EXPRESSION1 | EXPRESSION2 - Expression EXPRESSION3 may consist of either EXPRESSION1 or EXPRESSION2. This is equivalent to the two statements

EXPRESSION3 ::= EXPRESSION1

EXPRESSION3 ::= EXPRESSION2

The full description of AIML syntax follows:

AIML\_FILE ::= **<aiml( version="AIML\_VERSION")>AIML</aiml>**

AIML\_VERSION ::= **0.9 | 1.0 | 1.1 | 2.0**

AIML ::= (CATEGORY\_EXPRESSION | TOPIC\_EXPRESSION)\*

TOPIC\_EXPRESSION ::=

**<topic name="PATTERN\_EXPRESSION">(CATEGORY\_EXPRESSION)+</topic>**

CATEGORY\_EXPRESSION ::= **<category><pattern>PATTERN\_EXPRESSION</pattern>**

**(<that>PATTERN\_EXPRESSION</that>)(<topic>PATTERN\_EXPRESSION</topic>)**

**<template>TEMPLATE\_EXPRESSION</template></category>**

PATTERN\_EXPRESSION ::= WORD | PRIORITY\_WORD | WILDCARD | SET\_STATEMENT |

PATTERN\_SIDE\_BOT\_PROPERTY\_EXPRESSION

PATTERN\_EXPRESSION ::= PATTERN\_EXPRESSION PATTERN\_EXPRESSION

with exactly one space " " between pattern expressions.

The definition of WORD is language dependent. For English, we generally acknowledge any combination from the regular expression `[a-zA-Z0-9]*` as an AIML word. The AIML preprocessor step called

normalization (described in detail below) converts an input sentence to a normalized form where punctuation has been removed, and each word consists of an element of [a-zA-Z0-9]\*.

WILDCARD ::= \* | \_

SET\_STATEMENT ::= <set>SET\_NAME</set>

SET\_NAME ::= WORD

PRIORITY\_WORD ::= \$WORD

PATTERN\_SIDE\_BOT\_PROPERTY\_EXPRESSION ::= <bot name="PROPERTY\_NAME"/> |

<bot><name>PROPERTY\_NAME</name></bot>

PROPERTY\_NAME ::= WORD

Now we turn to the AIML template, which has a hierarchical structure:

TEMPLATE\_EXPRESSION ::= TEXT | TAG\_EXPRESSION | (TEMPLATE\_EXPRESSION)\*

TEXT is any ordinary English text consisting of any character except "<" and ">", which must be specified as &lt; and &gt; respectively.

NORAMLIZED\_TEXT is any TEXT that has been normalized by the AIML preprocessor. The exact normalization substitutions are left up to the botmaster.

TAG\_EXPRESSION ::=

RANDOM\_EXPRESSION |

CONDITION\_EXPRESSION |

SRAI\_EXPRESSION |

SRAIX\_EXPRESSION |

SET\_PREDICATE\_EXPRESSION |

GET\_PREDICATE\_EXPRESSION |

MAP\_EXPRESSION |

BOT\_PROPERTY\_EXPRESSION |

DATE\_EXPRESSION |

THINK\_EXPRESSION |

EXPLODE\_EXPRESSION |

NORMALIZE\_EXPRESSION |

DENORMALIZE\_EXPRESSION |

FORMAL\_EXPRESSION |

UPPERCASE\_EXPRESSION |

LOWERCASE\_EXPRESSION |

SENTENCE\_EXPRESSION |

PERSON\_EXPRESSION |

PERSON2\_EXPRESSION |

GENDER\_EXPRESSION |

SYSTEM\_EXPRESSION |

STAR\_EXPRESSION |

THATSTAR\_EXPRESSION |

TOPICSTAR\_EXPRESSION |

THAT\_EXPRESSION |

REQUEST\_EXPRESSION |

RESPONSE\_EXPRESSION |

LEARN\_EXPRESSION |

INTERVAL\_EXPRESSION |

<sr/> | <id/> | <vocabulary/> | <program/>

RANDOM\_EXPRESSION ::= <random>(<li>TEMPLATE\_EXPRESSION</li>)+</random>

CONDITION\_ITEM\_COMPONENT ::= <name>TEMPLATE\_EXPRESSION</name> |

<value>TEMPLATE\_EXPRESSION</value> | <loop/> | TEMPLATE\_EXPRESSION

CONDITION\_ITEM\_EXPRESSION ::= <li( CONDITION\_ATTRIBUTES)\*>

(CONDITION\_ITEM\_COMPONENT)\*</li>

CONDITION\_ATTRIBUTES ::= (name="NAME") | (value="NORMALIZED\_TEXT")

CONDITION\_EXPRESSION ::=

<condition( CONDITION\_ATTRIBUTES)>(CONDITION\_ITEM\_EXPRESSION)\*</condition>

SRAI\_EXPRESSION ::= <srai>TEMPLATE\_EXPRESSION</srai>

SRAIX\_EXPRESSION ::= <sraix( SRAIX\_ATTRIBUTES)\*>TEMPLATE\_EXPRESSION</sraix> |

```

<sraix>(SRAIX_ATTRIBUTE_TAGS)*TEMPLATE_EXPRESSION</sraix>
SRAIX_ATTRIBUTES ::= host="HOSTNAME" | botid="BOTID" | hint="TEXT" | apikey="APIKEY" |
service="SERVICE"
SRAIX_ATTRIBUTE_TAGS ::= <host>TEMPLATE_EXPRESSION</host> |
<botid>TEMPLATE_EXPRESSION</botid> | <hint>TEMPLATE_EXPRESSION</hint> |
<apikey>TEMPLATE_EXPRESSION</apikey> | <service>TEMPLATE_EXPRESSION</service>
GET_PREDICATE_EXPRESSION ::= <get name="WORD"/> |
<get><name>TEMPLATE_EXPRESSION</name></get> | <get var="WORD"> | <get><var>WORD</var>
</get>
SET_PREDICATE_EXPRESSION ::= <set name="WORD">TEMPLATE_EXPRESSION</set> |
<set><name>TEMPLATE_EXPRESSION</name>TEMPLATE_EXPRESSION</set> | <set
var="WORD">TEMPLATE_EXPRESSION</set> |
<set><var>TEMPLATE_EXPRESSION</var>TEMPLATE_EXPRESSION</set>
MAP_EXPRESSION ::= <map name="WORD">TEMPLATE_EXPRESSION</map> |
<map><name>TEMPLATE_EXPRESSION</name>TEMPLATE_EXPRESSION</map>
BOT_PROPERTY_EXPRESSION ::= <bot name="PROPERTY"/> |
<bot><name>TEMPLATE_EXPRESSION</name></bot>
DATE_EXPRESSION ::= <date( DATE_ATTRIBUTES)*/> | <date>(DATE_ATTRIBUTE_TAG)</date>
DATE_ATTRIBUTES ::= (format="LISP_DATE_FORMAT") | (jformat="JAVA DATE FORMAT")
DATE_ATTRIBUTE_TAG ::= <format>TEMPLATE_EXPRESSION</format> |
<jformat>TEMPLATE_EXPRESSION</jformat>
INTERVAL_EXPRESSION ::= <interval>(DATE_ATTRIBUTE_TAGS)<style>(TEMPLATE_EXPRESSION)
</style><from>(TEMPLATE_EXPRESSION)</from><to>(TEMPLATE_EXPRESSION)</to></interval>
THINK_EXPRESSION ::= <think>TEMPLATE_EXPRESSION</think>
EXPLODE_EXPRESSION ::= <explode>TEMPLATE_EXPRESSION</explode>
NORMALIZE_EXPRESSION ::= <normalize>TEMPLATE_EXPRESSION</normalize>
DENORMALIZE_EXPRESSION ::= <denormalize>TEMPLATE_EXPRESSION</denormalize>
PERSON_EXPRESSION ::= <person>TEMPLATE_EXPRESSION</person>
PERSON2_EXPRESSION ::= <person2>TEMPLATE_EXPRESSION</person2>
GENDER_EXPRESSION ::= <gender>TEMPLATE_EXPRESSION</gender>
SYSTEM_EXPRESSION ::=
    <system( TIMEOUT_ATTRIBUTE)>TEMPLATE_EXPRESSION</system> |
    <system><timeout>TEMPLATE_EXPRESSION</timeout></system>
TIMEOUT_ATTRIBUTE ::= timeout="NUMBER"
STAR_EXPRESSION ::= <star( INDEX_ATTRIBUTE)*/> | <star>
<index>TEMPLATE_EXPRESSION</index></star>
INDEX_ATTRIBUTE ::= index="NUMBER"
THATSTAR_EXPRESSION ::= <thatstar( INDEX_ATTRIBUTE)*/> | <thatstar>
<index>TEMPLATE_EXPRESSION</index></thatstar>
TOPICSTAR_EXPRESSION ::= <topicstar( INDEX_ATTRIBUTE)*/> | <topicstar>
<index>TEMPLATE_EXPRESSION</index></topicstar>
THAT_EXPRESSION ::= <that( THAT_INDEX)*/> | <that><index>TEMPLATE_EXPRESSION</index>
</that>
THAT_INDEX ::= index="NUMBER,NUMBER"
REQUEST_EXPRESSION ::= <request( INDEX_ATTRIBUTE)*/> | <request>
<index>TEMPLATE_EXPRESSION</index></request>
RESPONSE_EXPRESSION ::= <response( INDEX_ATTRIBUTE)*/> | <response>
<index>TEMPLATE_EXPRESSION</index></response>
LEARN_EXPRESSION ::= <learn>LEARN_CATEGORY_EXPRESSION</learn> |
<learnf>LEARN_CATEGORY_EXPRESSION</learnf>
LEARN_CATEGORY_EXPRESSION ::=
    <category><pattern>LEARN_PATTERN_EXPRESSION</pattern>
    (<that>LEARN_PATTERN_EXPRESSION</that>)(<topic>LEARN_PATTERN_EXPRESSION</topic>)
    <template>LEARN_TEMPLATE_EXPRESSION</template></category>
EVAL_EXPRESSION ::= <eval>TEMPLATE_EXPRESSION</eval>

```

```

LEARN_PATTERN_EXPRESSION ::= PATTERN_EXPRESSION | EVAL_EXPRESSION
LEARN_PATTERN_EXPRESSION ::= (LEARN_PATTERN_EXPRESSION)+
LEARN_TEMPLATE_EXPRESSION ::= TEXT | TAG_EXPRESSION | EVAL_EXPRESSION
LEARN_TEMPLATE_EXPRESSION ::= (LEARN_TEMPLATE_EXPRESSION)*

```

## 5. AIML Pattern Language

AIML patterns are made up of words, wildcards, AIML set expressions, and bot properties.

A word is any sequence of characters output by the normalization pre-processor that does not contain a space. The space character is reserved to indicate a space between words, as it does in many human languages including English. Exactly which characters are allowed in normalization depends on the botmaster's choice of normalization substitutions and the input language, but generally the idea with normalization is:

- Remove punctuation
- Expand contractions
- Correct a few common spelling mistakes
- Ensure one space between words

So "Hello", "123", "HaveFun" are normalized words but "can't", "1.23", and "Have-Fun" are not. Some AIML applications that require the bot to have knowledge of the original punctuation include normalization substitutions so that for example "," becomes "comma", "-" becomes "dash" and "." becomes "point".

One way to process inputs from languages like Japanese and Chinese that do no separate words with spaces is to place an implicit space between each character and treat each one as a "word".

Pre process the input

日本の伝統

into

日 本 の 伝 統

and use patterns like

<pattern>\* の 伝 統</pattern>

AIML 2.0 includes some new wildcards and pattern-side expressions.

## A. Zero or more words wildcards

The AIML 1.0 wildcards \* and \_ are defined so that they match one or more words. AIML 2.0 introduces two new wildcards, ^ and #, defined to match zero or more words. As a shorthand description, we refer to these as "zero+ wildcards".

Both ^ and # are defined to match 0 or more words. The difference between them is the same as the difference between \* and \_. The # matching operator has highest priority in matching, followed by \_, followed by an exact word match, followed by ^, and finally \* has the lowest matching priority.

When defining a zero+ wildcard it is necessary to consider what the value of <star/> (as well as <thatstar/> and <topicstar/>) should be when the wildcard match has zero length. In AIML 2.0 we leave this up to the botmaster. Each bot can have a global property named nullstar which the botmaster can set to "", "unknown", or any other value.

Examples:

```

<category><pattern>SHARPTTEST #</pattern>
<template>#star = <star/></template>
</category>

<category><pattern>SHARPTTEST # TEST</pattern>
<template>#star = <star/></template>
</category>

<category><pattern># KEYWORD #</pattern>
<template>Found KEYWORD</template>
</category>

<category><pattern>^ CARETTEST</pattern>
<template>^star = <star/></template>
</category>

```

Sample dialog:

Human: sharptest

Robot: #star = unknown



Human: keyword  
 Robot: Found KEYWORD  
 Human: sharptest foo  
 Robot: #star = foo  
 Human: sharptest foo bar test  
 Robot: #star = foo bar  
 Human: xyz abc carettest  
 Robot: ^star = xyz abc  
 Human: carettest  
 Robot: ^star = unknown  
 Human: keyword  
 Robot: Found KEYWORD  
 Human: abc def keyword ghi jkl  
 Robot: Found KEYWORD  
 Human: abc keyword  
 Robot: Found KEYWORD  
 Human: keyword def  
 Robot: Found KEYWORD

#### B. \$ operator

In some cases it is desirable to make an exact word match have higher priority than `_`.

For example,

The category

```
<category>
<pattern>_ ALICE</pattern>
<template><sr/></template>
</category>
```

is useful for removing the bot's name, Alice, from many queries such as "Tell me the time, Alice" and "What is your favorite color, Alice?". The `<sr/>` simplifies these inputs to "Tell me the time" and "What is your favorite color" respectively. But the category breaks down for other inputs like "Who is Alice?" which we wouldn't want to reduce to just "Who is".

Using the \$ operator we can add the category

```
<category>
<pattern>$WHO IS ALICE</pattern>
<template>I am Alice.</template>
</category>
```

so that the input "Who is Alice?" matches this category and not the one with `<pattern>_ ALICE</pattern>`.

The \$ indicates that the word has higher matching priority than `_`.

## AIML Sets

A pattern in AIML 2.0 may contain an expression referring to an AIML set. If the botmaster has defined a set named "color" of color names, then the expression

```
<set>color</set>
```

can match any member of this set.

## Examples of valid AIML patterns

The following are examples of valid AIML patterns

```
<pattern>*</pattern>
<pattern>HOW ARE YOU</pattern>
<pattern>How are you</pattern>
<pattern>HoW aRe YoU</pattern> -- AIML patterns are case invariant
<pattern><set>color</set></pattern>
<pattern><SET>COLOR</SET></pattern>
<pattern>I LIKE <set>color</set></pattern>
```

```

<pattern>_ THANK YOU</pattern>
<pattern>_ MUSIC *</pattern>
<pattern># MUSIC #</pattern>
<pattern>I LIKE # MUSIC</pattern>
<pattern>_ * _ * _</pattern> -- may not be useful but it is valid

```

### Examples of invalid patterns:

The following are not valid AIML patterns

```

<pattern></pattern> -- no concept of a blank pattern
<pattern>How are you?</pattern> -- no punctuation in AIML patterns
<pattern>I LIKE*</pattern> -- wildcard should have a space separating it from word
<pattern><set>*</set></pattern> -- no wildcard in set name
<pattern>_ * _ * _</pattern> -- wildcards should have spaces separating them

```

## 6. AIML Semantics

This section explains the semantics of each AIML tag.

### a. <aiml> tag

The <aiml> tag wraps the contents of an AIML file.

Example:

```

<?xml version="1.0" encoding="UTF-8"?>
<aiml version="2.0">
<category><pattern><SET>COLOR</SET></pattern>
<template><star/> is a color.</template>
</category>
</aiml>

```

The AIML file is an XML file and so may also have an optional header like

```
<?xml version="1.0" encoding="UTF-8"?>
```

however the definition of the XML header is outside the scope of AIML 2.0. The <aiml> tag wraps the AIML contents.

### b. <topic name="TOPIC"> tag

The <topic> tag wraps a collection of categories that share the same topic pattern.

Example:

```

<topic name="ASKING TO ADD NEW CONTACTNAME">
<category>
<pattern>YES</pattern><that>WOULD YOU LIKE TO ADD * AS A CONTACT</that>
<template>
<think><set name="topic">unknown</set>
<set name="contactid"><srai>NEWCONTACTID</srai></set>
<set name="displayname"><get name="contactname"/></set>
</think>
<srai>LEARN CONTACTID <get name="contactid"/> DISPLAYNAME <get name="displayname"/></srai>
I've saved <get name="contactname"/> to your contacts.
<srai>RESUMEACTION <get name="modecom"/></srai>
</template>
</category>
<category>
<pattern>*</pattern><that>WOULD YOU LIKE TO ADD * AS A CONTACT</that>
<template>
<think><set name="topic">unknown</set></think>
<srai>CONTACTFINALIZE</srai> <srai><star/></srai>
</template>
</category>
</topic>

```

In this example, the first category has input pattern YES and the second category has input pattern \*. Both categories have the same that pattern, WOULD YOU LIKE TO ADD \* AS A CONTACT, and the same topic pattern, ASKING TO ADD NEW CONTACTNAME.

In AIML 2.0 the topic pattern may also be defined inside a category. That is, a category has an input pattern

specified by the `<pattern>` tag, a that pattern specified by the `<that>` tag, and a topic pattern specified by the `<topic>` tag. If either `<that>` or `<topic>` is omitted, the corresponding pattern is defined as `*` by default.

### c. `<category>` tag

The basic unit of knowledge in AIML is a category. The `<category>` tag always contains an input `<pattern>` and a response `<template>`. Optionally it may also contain a `<that>` pattern and a `<topic>` pattern. If either of `<that>` or `<topic>` is omitted, the AIML interpreter assigns the corresponding pattern a value of `*`.

Example:

```
<category>
<pattern>HI</pattern>
<template>Hi there!</template>
</category>
```

### d. `<pattern>` tag

The `<pattern>` tag specifies the input pattern. The contents of the pattern tag are defined in the AIML Pattern Language section.

### e. Pattern-side `<set>`

A `<set>` tag in a `<pattern>` expression denotes an AIML Set, a collection of words or phrases (sequences of words) that can be matched by this part of the input pattern. The `<set>` tag contains the name of the AIML Set. The exact representation of the set is left up to the interpreter. The set should contain normalized items that can be matched by normalized inputs.

AIML matching treats an AIML set much like a wildcard (`*` or `_`). The `<set>` expression matches one or more words. The `<set>` match has higher priority than `*`, but lower priority than an exact word match. See the section AIML Pattern Matching for more details.

Whichever words match the `<set>` expression may be retrieved on the template-side with `<star/>`. If there are more than one `<set>` elements in a pattern, the matching word sequences may be retrieved with `<star index="1"/>`, `<star index="2"/>` and so on.

Similarly, any sequences matching a `<set>` expression in a that-pattern or topic-pattern may be accessed on the template-side with `<thatstar/>` and `<topicstar/>`.

Examples:

```
<category>
<pattern>I LIKE <set>color</set></pattern>
<template><star/> is a nice color.</template>
</category>

<category>
<pattern>CALL <set>number</set></pattern>
<template><srai>DIAL <star/></srai></template>
</category>

<category>
<pattern><set>name</set></pattern>
<template><star/> is a name.</template>
</category>
```

Sample dialogs:

Human: I like blue.

Robot: Blue is a nice color.

Human: Call 5551234

Robot: Now dialing 5551234

Human: Joseph

Robot: Joseph is a name.

See the companion document [Sets and Maps in AIML 2.0](#)

### f. Pattern-side `<bot name="property"/>`

The bot property tag may appear in a pattern expression. It is equivalent to a word in pattern matching.

Example:

```
<category>
<pattern>ARE YOU <bot name="name"/></pattern>
<template>Yes, I am.</template>
</category>
```

### g. `<that>`

The `<that>` tag contains an AIML pattern. Like `<pattern>` and `<topic>`, `<that>` may contain any valid AIML

pattern elements, including words, wildcards, `<set>` and `<bot>` elements.

The purpose of `<that>` is to match the bot's last utterance, specifically, the last sentence of the last response.

Typically `<that>` plays an important role in question answering. If the client says "Yes", the bot should remember what question it asked to make the client say "Yes", so that it can put together the affirmative response with the question, in order to formulate a reply.

Examples:

```
<category><pattern>YES</pattern><that>ARE YOU TIRED</that>
<template>Maybe you should get some rest. I will still be here later.</template>
</category>
<category><pattern>NO</pattern><that>CAN YOU HEAR ME</that>
<template>Try adjusting the media volume on your device Settings.</template>
</category>
<category><pattern>*</pattern><that>WHAT IS YOUR NAME</that>
<template><srai>MY NAME IS <star/></srai></template>
</category>
```

#### h. `<topic>`

AIML 2.0 migrates the `<topic>` tag from a category wrapper to a tag inside a category. For backwards compatibility, the category wrapper tag (see subsection b. **`<topic name="TOPIC">` tag**) continues to be permitted in AIML 2.0. But the `<topic>` tag around a category was always confusing, because the AIML pattern matcher builds a pattern path by appending the input pattern, that pattern and topic pattern in that order. Having the `<topic>` tag around a category suggests that the order might have been: topic pattern, input pattern, that pattern, which was not the case.

The `<topic>` tag in a category, like `<pattern>` and `<that>`, contains a valid AIML pattern. Unlike the `<topic name="TOPIC">` tag described in subsection b, the `<topic>` tag inside a category omits the name attribute and simply encloses the topic pattern in a pair of tags.

Examples:

```
<category>
<pattern>*</pattern>
<topic>TRAVEL</topic>
<template>Have you been to
<random>
<li>Rome</li>
<li>London</li>
<li>Paris</li>
</random>?
</template>
</category>
<category>
<pattern>_</pattern><that>WHAT IS YOUR MESSAGE TO *</that>
<topic>ASKING MESSAGEBODY</topic>
<template>
<think>
<set name="topic">unknown</set>
<set name="messagebody"><star/></set>
</think>
<srai>RESUMEACTION <get name="modecom"/></srai>
</template>
</category>
```

#### i. `<template>`

The `<template>` contains the AIML response. In its simplest form, the response consists of plain text. In general however, the `<template>` contains what is in effect a mini computer program, written with AIML tags, that computes a response. Generally, a `<template>` contains a mix of plain text and AIML tags.

Because the `<template>` may contain the `<srai>` tag, this computation may activate other AIML categories, and evaluate their `<template>` responses to build a response recursively.

Examples:

```
<category><pattern>YOU ARE HELPFUL</pattern>
<template>I like to help people.</template>
```

```

</category>
<category><pattern>NO YOU *</pattern>
<template><srai>NO</srai> <srai>YOU <star/></srai></template>
</category>
<category><pattern>NAME</pattern>
<template>
<random>
<li>I am</li>
<li>Call me</li>
<li>My name is</li>
<li>I am called</li>
<li>People call me</li>
<li>You can call me</li>
</random>
<condition name="customname">
<li value="unknown"><bot name="name"/>.</li>
<li><get name="customname"/>.</li>
</condition>
</template>

```

This discussion of the `<template>` tag completes the tour of `<category>` subelements. To summarize: every category has a `<pattern>` and a `<template>`, and optionally a `<that>` and/or `<topic>`. The remaining AIML tags discussed in the subsections below are all subtags of `<template>`.

#### j. `<random>`

The purpose of the `<random>` tag is to allow the bot to select one of a list of responses randomly. The distribution of selections should be random uniform: no response has higher probability of selection than the other choices. The `<random>` tag uses the `<li>` subtag to indicate selections. The `<li>` items may contain any valid `<template>` expression.

The AIML interpreter replaces the `<random>` tag and its contents with the value of the evaluated `<li>` expression.

Example:

```

<category><pattern>PURPOSE</pattern>
<template>
<random>
<li>I'm here to help you in any way I can.</li>
<li>I am a mobile virtual assistant, ready to do what I can for you.</li>
<li>I'm here to help.</li>
</random>
</template>
</category>

```

Sample dialog:

Human: Purpose?

Robot: I'm here to help

Human: Purpose:

Robot: I am a mobile virtual assistant, ready to do what I can for you.

#### k. `<condition name="predicate" value="v">` and `<condition var="varname" value="v">`

The `<condition>` tag has three basic forms. described in this and the next two subsections. The first type is known as the "one-shot condition" and contains both a predicate name and a value attribute. If the named predicate value equals the given value, the AIML interpreter replaces the `<condition>` tag and its contents with the result of evaluating those contents. If the values are not equal, the `<condition>` expression is replaced with a blank.

AIML 2.0 adopts one of the Pandorabots extensions to the `<condition>` tag, where a value of `*` indicates that the predicate is bound to something. If a predicate `p` has not been set by a previously evaluated `<set name="p">` tag (nor initialized by the interpreter), then it is called "unbound". A predicate with an assigned value is "bound".

The expression `<condition name="p" value="*"> X</condition>` evaluates to `X` provided that `p` is a bound predicate. Otherwise, it evaluates to a blank. This `*`-value notation applies to all three forms of the `<condition>` tag.

In AIML 2.0 the name and/or value attribute values may be specified in subtags instead of XML attribute values. The subtags have the same names as the original attribute names.

```
<condition name="predicate" value="v">X</condition>,
<condition name="predicate"><value>v</value>X</condition>,
<condition value="v"><name>predicate</name>X</condition>, and
<condition><name>predicate</name><value>v</value>X</condition>
```

are all equivalent.

**Example:**

```
<category>
<pattern>DIALNUMBER UNKNOWN *</pattern>
<template>
<think>
<set name="dialnumber"><srai>DIALNUMBER MOBILE <star/></srai></set>
<condition name="dialnumber" value="unknown">
<set name="dialnumber"><srai>DIALNUMBER HOME <star/></srai></set>
</condition>
<condition name="dialnumber" value="unknown">
<set name="dialnumber"><srai>DIALNUMBER WORK <star/></srai></set>
</condition>
<condition name="dialnumber" value="unknown">
<set name="dialnumber"><srai>DIALNUMBER CUSTOM <star/></srai></set>
</condition>
</think>
<get name="dialnumber"/></template>
</category>
```

**I. <condition name="predicate"> or <condition var="varname"> and <li value="v">**

The second form of the condition tag contains only the predicate attribute. In AIML 2.0, this attribute may be specified in a subtag:

`<condition name="predicate">...</condition>` is equivalent to

```
<condition><name>predicate</name>...</condition>
```

This form of `<condition>` contains a list of items specified by the `<li>` tag. The interpreter checks each list item to see if the predicate value equals the specified value in the associated list item, and if they are equal, replaces the `<condition>` tag and its contents by the result of evaluating the list item contents.

Each list item except optionally the last has the form

```
<li value="v">X</li>
```

or

```
<li><value>X</li>
```

The interpreter checks the list items one at a time to see if an equality condition holds. For the first list item containing equal values, the interpreter evaluates the result of the associated list item and replaces the contents of the `<condition>` tag and its contents with that evaluated result.

A `<condition name="predicate">` may optionally contain a final list item of the form

```
<li>X</li>
```

with no specified value. If the interpreter is unable to find a true equality condition for the previous list of `<condition>` items, and the list contains this last default item, then the interpreter replaces the `<condition>` tag and its contents with the result of evaluating the contents of this final item. If the default item is omitted and none of the other items are selected, the result of the `<condition name="predicate">` tag is blank.

Like the one-shot `<condition>`, the `<condition name="predicate">` tag may use `*` for the value attribute to test whether a predicate is bound or not.

**Examples:**

```
<category>
<pattern>HE IS GOING TO *</pattern>
<template>
<condition name="he">
<li value="who">Who is he?</li>
<li><get name="he"/> is going to?</li>
</condition>
</template>
</category>
```

```

<category>
<pattern>I AM *</pattern>
<template>
<think>
<set name="isanumber"><srai>ISANUMBER <star/></srai></set>
<set name="isaname"><srai>ISANAME <star/></srai></set>
</think>
<condition name="isanumber">
<li value="true"><srai>MY AGE IS <star/></srai></li>
<li>
  <condition name="isaname">
    <li value="true"><srai>MY NAME IS <star/></srai></li>
    <li><srai>IAMRESPONSE</srai>
  </li>
</condition>
</li>
</condition>
</template>
</category>
<category><pattern>IS * EQUALTO *</pattern>
<template>
<think><set name="temp"><star/></set></think>
<condition name="temp">
<li><value><star index="2"/></value> true</li>
<li>false</li>
</condition>
</template>
</category>

```

**m. <condition> and <li name="predicate" value="v"> or <li var="varname" value="v">**

The final form of the <condition> tag is the most general. It contains list items, and each list item (except optionally the last) specifies a predicate name and a value.

As with the first two forms of <condition>, the predicate name and value may be specified in subtags as well as in attribute values:

```

<li name="predicate" value="v">X</li>
<li name="predicate"><value>v</value>X</li>,
<li value="v"><name>predicate</name>X</li>, and
<li><name>predicate</name><value>v</value>X</li> are all equivalent.

```

The interpreter checks the list items one at a time to see if an equality condition holds. For the first list item containing equal values, the interpreter evaluates the result of the associated list item and replaces the contents of the <condition> tag and its contents with that evaluated result.

A general <condition> may optionally contain a final list item of the form

```
<li>X</li>
```

with no specified value. If the interpreter is unable to find a true equality condition for the previous list of <condition> items, and the list contains this last default item, then the interpreter replaces the <condition> tag and its contents with the result of evaluating the contents of this final item. If the default item is omitted and none of the other items are selected, the result of the <condition> tag is blank.

Like the one-shot <condition name="predicate" value="v"> and the <condition name="predicate">, the general <condition> tag may use \* for the value attribute to test whether a predicate is bound or not.

## Local variables

Like <set> and <get> in AIML 2.0, the <condition> tags may refer to a local variable instead of a predicate. If the attribute in a <condition> expression in some template is var instead of name, then the condition statement looks for a local variable within the scope of the template.

Example:

```

<category><pattern>IS _ EQUALTO *</pattern>
<template>

```

```

<think><set var="star"><star/></set>
</think>
<condition var="star">
<li><value><star index="2"/></value>true</li>
<li>false</li>
</condition>
</template>
</category>

```

Here the var indicates that the variable star has scope limited to this category. This lets the botmaster reuse common names such as "star" in many different categories, without worrying about saving this value in a global predicate value.

#### n. <loop/>

AIML 2.0 provides for a <loop/> tag in a condition list item <li> element. If the activated <li> element contains a <loop/> tag, then the condition statement is re-evaluated until it reaches another <li> item without <loop/>.

When an <li> element containing <loop/> is evaluated, the interpreter replaces the <loop/> tag with an empty string.

Each time <loop/> returns and the <condition> tag is re-evaluated, any text resulting from the evaluation of the <li> item is appended to a concatenated string, so that the finally the <condition> element is replaced with the appended, evaluated list items.

```

<category>
<pattern>COUNT TO <set>number</set></pattern>
<template>
<think><set name="count">0</set></think>
<condition name="count">
<li><value><star/></value></li>
<li>
<set name="count"><map><name>successor</name><get name="count"/></map></set> <loop/>
</li>
</condition>
</template>
</category>
<category>
<pattern>NTH <set>number</set> *</pattern>
<template>
<think>
<set name="nth"><star/></set>
<set name="count">1</set>
<set name="letters"><explode><star index="2"/></explode></set>
</think>
<condition>
<li><name>letters</name><value>undefined</value>
<star index="2"/> has only <map><name>predecessor</name><get name="count"/></map> letters.</li>
<li><name>count</name><value><get name="nth"/></value>
The <map><name>number2ordinal</name><star/></map> letter is <srai>FIRSTLETTER <get
name="letters"/></srai></li>
<li>
<think>
<set name="count"><map><name>successor</name><get name="count"/></map></set>
<set name="letters"><srai>REMAININGLETTERS <get name="letters"/></srai></set>
</think>
<loop/>
</li>
</condition>
</template>
</category>

```

#### Sample dialogs:

Human: Count to 14



Robot: 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
 Human: what is the first letter of dog?  
 Robot: The First letter is d.  
 Human: what is the seventh letter of church?  
 Robot: church has only 6 letters.  
 Human: what is the 22nd letter of the alphabet?  
 Robot: The Twenty second letter is V  
 Human: what is the eleventh letter in greenhouse?  
 Robot: greenhouse has only 9 letters.  
 Human: what is the 7th letter within greenhouse?  
 Robot: The Seventh letter is o

#### **o. <sr> and <sr/>**

There has never been any agreement over what the "sr" stands for in the <sr> tag. This is because the tag has so many applications in AIML. At different times, we describe <sr> as Symbolic Reduction, Simple Recursion, Syntactic Rewrite or Stimulus-Response.

When the interpreter encounters an <sr> tag, it evaluates the contents and converts the result to normalized form. The interpreter then feeds these normalized, evaluated contents back into the pattern matcher to find a matching category. When the matching category is found, the interpreter evaluates its template and replaces the <sr> with the result.

A very common expression in AIML is

```
<sr><star></sr>
```

where the <sr> is applied to whatever matched the first wildcard \* or \_ (or in AIML 2.0, a pattern-side <set>). Because the botmaster may end up writing this expression so often, AIML includes an abbreviation <sr/> defined as

```
<sr/> = <sr><star></sr>.
```

The best way to understand the recursive action of the AIML <sr> tag is by example.

Client: You may say that again Alice.

Robot: Once more? "that."

The robot has no specific response to the pattern "You may say that again Alice." Instead, the robot builds its response to the client input in four steps. This simple sentence activated a sequence of four categories linked by <sr> tags. The robot constructed the reply "Once more? "that"" recursively as each subsentence triggered the next matching pattern.

In this example the processing proceeds in four steps, because each of the first three steps evokes another symbolic reduction.

Step	normalized input	matching pattern	template	response
1.	YOU MAY SAY THAT AGAIN ALICE	_ <bot name="name"/>	<sr/>	
2.	YOU MAY SAY THAT AGAIN	_ AGAIN	Once more? <sr/>	Once More?
3.	YOU MAY SAY THAT	YOU MAY *	<sr/>	Once More?
4.	SAY THAT	SAY *	"<person/>"	Once More? "that".

In step 1, the patterns with "\_" match first because they are last in alphabetical order. The order of the matches depends on this alphabetical ordering of patterns. ALICE always matches suffixes with "\_" before prefixes with "\*". Whatever matches either wild-card symbol becomes the value of <star/>.

Steps 1 through 3 illustrate the common AIML templates that use the abbreviated <sr/> tag. (Remember, <sr/> = <sr><star></sr> ). The categories with the patterns "\_ <name/>" and "YOU MAY \*" simply reduce the sentence to whatever matches the "\*", as illustrated by steps 1 and 3.

Some AIML templates in ALICE combine the <sr/> with an ordinary text response, as step 2 with the pattern "\_ AGAIN". The phrase "Once more?" becomes part of any reply ending in "AGAIN".

The category in step 4 with "SAY \*" is a default that often produces logically correct but amusing dialogue:

Client: Say hello in Swedish.

Robot: "Hello in Swedish."

or as in this case:

Client: Say that.

Robot: "that."

## Many patterns, one reply

The most common use of `<srai>` is to map two, or more, patterns to the same response:

```
<category>
  <pattern>P</pattern>
  <template>
    <srai>Q</srai>
  </template>
</category>
<category>
  <pattern>Q</pattern>
  <template>R</template>
</category>
```

An input matching either pattern, P or Q, gets the same response R.

To show a more concrete example: the input "Hello" should have an appropriate response like "Hi there!". But we can expand the inputs generating this response to include all the common variations of "Hello":

```
<category>
  <pattern>HI</pattern>
  <template>
    <srai>HELLO</srai>
  </template>
</category>
<category>
  <pattern>HOWDY</pattern>
  <template>
    <srai>HELLO</srai>
  </template>
</category>
<category>
  <pattern>HALLO</pattern>
  <template>
    <srai>HELLO</srai>
  </template>
</category>
<category>
  <pattern>HI THERE</pattern>
  <template>
    <srai>HELLO</srai>
  </template>
</category>
<category>
  <pattern>HELLO</pattern>
  <template>Hi there!</template>
</category>
```

As the following example shows, we can use the `<sr/>` tag as an abbreviation for `<srai><star/></srai>`. This category creates a compound response to both "hello" and whatever matches the wildcard \*:

```
<category>
  <pattern>HELLO *</pattern>
  <template>
    <srai>HELLO</srai> <sr/>
  </template>
</category>
```

## Symbolic reductions and state

The tag `<that>` in AIML introduces one dimension of "dialogue state" into the robot response. The value of `<that>` is whatever the robot said before that provoked the current client input. The inputs "yes" and "no" are two of the most common human queries. But a careful analysis of the dialogues shows that most of the time, people say "yes" or "no" to only a small set of questions asked by the robot.

```
<category>
  <pattern>NO</pattern>
  <that>I UNDERSTAND</that>
  <template>
    <srai>YOU DO NOT UNDERSTAND</srai>
  </template>
</category>
```

illustrates the use of `<that>` with `<srai>`. The client input matches simply "No". What did the robot say that made the client say no? If it was "I understand." then this category formulates a response with

```
<srai>YOU DO NOT UNDERSTAND</srai>
```

which in turn activates another category with the pattern

```
<pattern>YOU DO NOT UNDERSTAND</pattern>
```

This category responds: "I do so understand. It all makes sense to my artificial mind."

## Summary of `<srai>`

The AIML `<srai>` tag simplifies and combines four important chat robot operations:

- Maps multiple patterns to the same response.
- Reduces a complex sentence structure to a simpler form.
- Diminishes the need for multiple-wildcard input patterns.
- Translates state-dependent inputs into simpler stimuli.

In some sense `<srai>` is a very low-level operation, but its simplicity captures a wide range of typical chat robot functions.

### p. `<sraix>`

`<sraix>` is a new AIML 2.0 tag designed to allow a bot to access external natural language web services and also other AIML bots. The `<sraix>` tag extends the concept of the classic AIML `<srai>` tag. The `<srai>` tag essentially allows a bot to reformulate an input and feed the revised input back to itself. The `<sraix>` tag feeds the reformulated input to another bot.

Recall the use of the classic `<srai>` in this example:

```
<category>
  <pattern>WHAT IS *</pattern>
  <template><srai>XFINND <star/></srai></template>
</category>
```

This category matches inputs starting with "What is". The purpose of the `<srai>` here is to rewrite the input as a sentence starting with the keyword XFINND. For example, "What is hermeneutics?" would be rewritten as "XFINND hermeneutics". There is another category with the pattern "XFINND \*" might contain a list of random responses to try to cover the fact the bot doesn't really know the answer, or it might contain some Javascript to try to access the information from a remote source. It has not been possible with AIML, until now, to directly query another bot or service for the response.

The `<sraix>` tag allows the botmaster to write categories that access other services. The "What is" category may now be written as:

```
<category>
  <pattern>WHAT IS *</pattern>
  <template><sraix>WHAT IS <star/></sraix></template>
</category>
```

In this case the input is not really reformulated at all. The template merely sends, in our example, "What is hermeneutics" to another bot. The other bot in the default case is the Pannous web service. In the general case, the bot could be a remote web service, or even another Pandorabot.

## `<sriax>` attributes

**No attribute**

The default case should access Pannous service (with what API key?).

**bot attribute**

`<sraix bot="username/botname">` - On a server with multiple bots created by multiple users, specify the destination bot by a combination of username and botname.

**limit attribute**

`<sraix limit="n">` - limit the response to the first n sentences received from the remote bot. For example `<sraix limit="5">` clips the response after the first 5 sentences. The default limit value is 3.

**service attribute**

`<sraix service="pannous">` - access a named service. We should register names of high-usage services like "pannous" for the convenience of the botmaster.

**apikey attribute**

`<sraix service="pannous" apikey="1234567">` - some services may require botmaster-specific API keys.

**botid attribute**

`<sraix botid="b69a1aa27e345abb">` - access a bot on the same server using its bot id.

**server attribute**

`<sraix server="www.pandorabots.com" botid="b69a1aa27e345abb">` - access another Pandorabot.

**hint attribute**

`<sraix hint="suggestion">` -- give the sraix processor a hint about which type of service to use.  
`hint="event"` -- means calendar or alarm event.

**default attribute**

`<sraix default="default reply">` makes "default reply" the response when `<sraix>` fails, instead of calling the SRAIXFAILED category.

Examples:

```
<category><pattern>WHAT IS THE WEATHER FORECAST FOR *</pattern>
<template><sraix>WHAT IS THE WEATHER FORECAST FOR <star/></sraix></template>
</category>
<category><pattern>FAVORITE *</pattern>
<template><sraix botid="844a85704e345a9a" host="callmom.pandorabots.com">FAVORITE <star/></sraix>
</template>
</category>
<category><pattern>SCHEDULEDESCRIPTION * SCHEDULEREQUEST *</pattern>
<template>
<think>
<set name="sraix"><sraix default="SCHEDULEFAILED" hint="event"><star index="2"/></sraix></set>
</think>
<condition name="sraix">
<li value="SCHEDULEFAILED">I'm unable to schedule that event.</li>
<li>I'll put that event on your calendar.
<oob><schedule><title><star/></title><description><lowercase><star index="2"/></lowercase>
</description><get name="sraix"/></schedule></oob>
</li>
</condition>
</template>
</category>
```

Sample dialog:

**Human:** weather in san francisco

**Robot:** San francisco It is currently mostly cloudy 60 Fahrenheit.

**Human:** what is the capital of texas

**Robot:** Capital of Texas usually refers to the Austin Texas the present capital.

However several other cities served as the capital of Texas During the period of the Republic of Texas 1836-1845 the capital of Texas moved from city to city.

**Human:** define arbitration

**Robot:** Arbitration a form of alternative dispute resolution ADR is a legal technique for the resolution of disputes outside the courts where the parties to a dispute refer it to one or more persons the arbitrators arbiters or arbitral tribunal by whose decision the award they agree to be bound. It is a resolution technique in which a third party reviews the evidence in the case and imposes a decision that is legally

binding for both sides and enforceable 1 Other forms of ADR include mediation 2 a form of settlement negotiation facilitated by a neutral third party and non binding resolution by experts. Arbitration is often used for the resolution of commercial disputes particularly in the context of international commercial transactions.

**Human:** who won the most olympic medals?

**Robot:** Michael Phelps has won the most medals in a career with 19 medals. 15 gold medals 2 silver and 2 bronze. He is from the United States of America and is a swimmer.

**Human:** do you know who phyllis diller is

**Robot:** Phyllis Diller July 17 1917 – August 20 2012 was an American actress and comedian. She created a stage persona of a wild haired eccentrically dressed housewife who made self deprecating jokes about her age and appearance her terrible cooking and a husband named Fang while pretending to smoke from a long cigarette holder. Diller's signature was her unusual laugh.

## Using <sraix> in AIML

Important Note:

To avoid the possibility of infinite recursion, you should include an AIML category with the pattern SRAIXFAILED. When the <sraix> operation is unable to connect to the remote bot, for example if the service is down or network connection lost, or if the remote bot is unable to provide an answer, Pandorabots will attempt to activate the category with pattern SRAIXFAILED. **It is extremely important not to include another <sraix> in the response template for SRAIXFAILED.**

**Example:**

```
<category>
<pattern>SRAIXFAILED</pattern>
<template>
<random>
<li>I am unable to answer.</li>
<li>I asked another robot, but he did not know.</li>
<li>Try asking me a different way.</li>
</random>
</template>
</category>
```

**Bad Example:**

```
<category>
<pattern>*</pattern>
<template><sraix><star/></sraix></template>
</category>
```

If this category exists and no SRAIXFAILED is found, the bot will go into infinite recursion, stopping when the interpreter reaches its recursion limit..

## <sraix> Example

We'll start with a simple example where we use <sraix> to answer "Who is..." questions.

We can write a simple AIML category to answer "Who is..." questions by contacting Pannous service.

```
<category>
<pattern>WHO IS *</pattern>
<template><sraix>WHO IS <star/></sraix></template>
</category>
```

This category will match inputs like “Who is Abraham Lincoln”, “Who is Alan Turing”, and “Who is Bob Marley”, all of which could be answered by Pannous.

This category by itself however will not match all the ways of asking “Who is X”, nor will all the inputs it matches be questions that Pannous can answer. As examples, consider:

(A). “Who is your mother?” - Matches “WHO IS \*”, but is really a personality question for the bot. (false positive match)

(B). “Do you know who Alan Turing is?” - Does not match “WHO IS \*”, but could be answered by Pannous. (false negative match)

To narrow the focus of the queries and resolve problem (A), the bot should have a number of other patterns and responses to answer personality questions, give opinions, and provide profile information. These include:

```
WHO IS YOUR MOTHER
WHO IS BETTER * OR *
WHO ARE YOUR FRIENDS
WHO IS MY GIRLFRIEND
WHO DO YOU LIKE BETTER * OR *
WHO IS YOUR PROGRAMMER
```

To increase the accuracy of matching for inputs that should be directed to Pannous, we can add categories with specific patterns like:

```
WHO IS THE PRESIDENT OF *
WHO WROTE *
WHO PLAYED *
WHO WON *
WHO INVENTED *
WHO IS THE LEAD SINGER OF *
```

To increase the variety of questions that can be asked and resolve problem (B), we need a set of reduction categories to capture various ways of phrasing “Who”-questions:

```
DO YOU KNOW WHO * IS → <srai>WHO IS <star/></srai>
WHO THE HELL IS * → <srai>WHO IS <star/></srai>
TELL ME WHO * IS → <srai>WHO IS <star/></srai>
SO WHO IS * → <srai>WHO IS <star/></srai>
```

#### q. **<set name="predicate">** and **<set var="variable">** (template-side)

The template-side **<set>** tag provides a way to set local variables, called predicates, that are specific to one client chatting with the bot. When evaluating a **<set>** tag, the interpreter processes the contents of the tag and stores the result as the value of the named predicate. The **<set>** expression is then replaced with this value, except in cases where the botmaster has specified the return value should be the predicate name instead of the value, as described below.

Like other template-side tags in AIML, the attribute for **<set>** may be specified in a subtag:

```
<set name="predicate">X</set> is equivalent to <set><name>predicate</name>X</set>
```

The botmaster may choose to have certain pronoun predicates, such as he, she and it, configured so that **<set>** returns the pronoun name rather than the value.

Given the category

```
<category>
<pattern>WHO IS LIONEL MESSI</pattern>
<template><set name="he">Lionel Messi</set> is a famous football star.</template>
</category>
```

the botmaster might prefer to have the response be

He is a famous football star.

rather than

Lionel Messi is a famous football star.

The exact method used to specify these predicate-name-return cases is left up to the interpreter.

A special case of **<set>** is **<set name="topic">**. “Topic” is a reserved predicate name for the AIML topic.

Whatever value the predicate topic has, is used in the AIML matching process and is significant when matching a category with a **<topic>** tag.

(For a description of the difference between the name and var attributes, see the following subsection on **<get var="variable">**).

#### r. **<get name="predicate"/>** and **<get var="variable"/>**

The complement to the template-side **<set>** tag is the **<get>** tag, which retrieves the value of a named

predicate. If the predicate has been set by a previously activated, the interpreter replaces the <set> tag with that saved value. Predicates that have not yet been set may have default values specified by the botmaster.

The format and means of specifying these default values is left up to the interpreter. There should also be a global default value for any predicates that have neither been set, nor have explicit default values.

The difference between the attributes name and var is like the distinction between global and local variables.

A var only has scope for the category template in which it is set. If we try to access a var value outside of this scope, it should return whatever global default value the botmaster defined for predicates. A var never has a default value, except this global default value.

Example:

```
<category><pattern>TEST VAR</pattern>
<template>
<think>
<set name="boundpredicate">some value</set>
<set var="boundvar">something</set>
</think>
TEST VAR:
unboundpredicate = <get name="unboundpredicate"/>.
boundpredicate = <get name="boundpredicate"/>.
unboundvar = <get var="unboundvar"/>.
boundvar = <get var="boundvar"/>.
<srai>TEST VAR SRAI</srai>
</template>
</category>
```

Sample dialog:

Robot: TEST VAR:

unboundpredicate = unknown.

boundpredicate = some value.

unboundvar = unknown.

boundvar = something.

TEST VAR SRAI:

unboundpredicate = unknown.

boundpredicate = some value.

unboundvar = unknown.

boundvar = unknown.

In this sample dialog, the first category with pattern TEST VAR contains a local variable called boundvar.

The value of boundvar is set to "something", and can be retrieved within that category by <get var="boundvar"/>. When the category activates another one through <srai> however, the value of boundvar is "unknown", the global default predicate value.

#### s. <map>

See the companion document [Sets and Maps in AIML 2.0](#)

#### t. <bot name="property"/>

Bot properties are like global constants for a bot. Once they are set by the botmaster, bot properties, unlike predicates, are not changed by conversational interaction. The bot properties are typically used for profile features such as the bot's name, age, gender, location, preferences and so on.

Like other tags in AIML 2.0, the attribute value of <bot name="property"/> may appear in a subtag:

<bot name="property"/> is equivalent to <bot><name>property</name></bot>.

When the AIML interpreter evaluates a <bot name="property"/> tag, it replaces the tag with the value of the named property. A global default value should exist for any property that has not been set by the botmaster.

#### u. <date format="f" jformat="j"/>

Pandorabots supports three extension attributes to the date element in templates:

```
locale
format
timezone
```

timzone should be an integer number of hours +/- from GMT and that locale is the iso language/country code pair e.g., en\_US, ja\_JP. Locale defaults to en\_US. The set of supported locales are:

```
af_ZA ar_OM da_DK en_HK es_CO es_PY fr_CA is_IS mt_MT sh_YU vi_VN
ar_AE ar_QA de_AT en_IE es_CR es_SV fr_CH it_CH nb_NO sk_SK zh_CN
ar_BH ar_SA de_BE en_IN es_DO es_US fr_FR it_IT nl_BE sl_SI zh_HK
ar_DZ ar_SD de_CH en_NZ es_EC es_UY fr_LU ja_JP nl_NL sq_AL zh_SG
ar_EG ar_SY de_DE en_PH es_ES es_VE ga_IE kl_GL nn_NO sr_YU zh_TW
ar_IN ar_TN de_LU en_SG es_GT et_EE gl_ES ko_KR no_NO sv_FI
ar_IQ ar_YE el_GR en_US es_HN eu_ES gv_GB kw_GB pl_PL sv_SE
ar_JO be_BY en_AU en_ZA es_MX fa_IN he_IL lt_LT pt_BR ta_IN
ar_KW bg_BG en_BE en_ZW es_NI fa_IR hi_IN lv_LV pt_PT te_IN
ar_LB bn_IN en_BW es_AR es_PA fi_FI hr_HR mk_MK ro_RO th_TH
ar_LY ca_ES en_CA es_BO es_PE fo_FO hu_HU mr_IN ru_RU tr_TR
ar_MA cs_CZ en_GB es_CL es_PR fr_BE id_ID ms_MY ru-UA uk-UA
```

format is a format string as given to the Unix strftime function:

<http://www.opengroup.org/onlinepubs/007908799/xsh/strftime.html>

You can include your own message in the format string, along with one or more format control strings. These format control strings tell the date function whether to print the date or time, whether to use AM or PM, a 24 hour clock or a 12 hour, abbreviate the day of the week or not, and so on. Some of the supported format control strings include:

```
%a Abbreviated weekday name
%A Full weekday name
%b Abbreviated month name
%B Full month name
%c Date and time representation appropriate for locale
%d Day of month as decimal number (01 - 31)
%H Hour in 24-hour format (00 - 23)
%I Hour in 12-hour format (01 - 12)
%j Day of year as decimal number (001 - 366)
%m Month as decimal number (01 - 12)
%M Minute as decimal number (00 - 59)
%p Current locale's A.M./P.M. indicator for 12-hour clock
%S Second as decimal number (00 - 59)
%U Week of year as decimal number, with Sunday as first day of week (00 - 53)
%w Weekday as decimal number (0 - 6; Sunday is 0)
%W Week of year as decimal number, with Monday as first day of week (00 - 53)
%x Date representation for current locale
%X Time representation for current locale
%y Year without century, as decimal number (00 - 99)
%Y Year with century, as decimal number
%Z Time-zone name or abbreviation; no characters if time zone is unknown
%% Percent sign
```

If you don't specify a format you'll just get the date using the default format for the particular locale.

timezone is the time zone expressed as the number of hours west of GMT.

If any of the attributes are invalid, it will fall back to the default behavior of <date/> (i.e. with no attributes specified)



To display the date and time in French using Central European time you would use:

```
<date locale="fr_FR" timezone="-1" format="%c"/>
```

You can also improve the specificity of common certain time and date related inquiries to the ALICE bot, as illustrated by the following dialogue fragment.

Human: what day is it  
 ALICE: Thursday.  
 Human: what month is it  
 ALICE: December.  
 Human: what year is this  
 ALICE: 2004.  
 Human: what is the date  
 ALICE: Thursday, December 02, 2004.

The `jformat` attribute relies on the Java Simple Date Format specification:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/text/SimpleDateFormat.html>

#### v. **<think>**

The purpose of the `<think>` tag is to provide a means for the interpreter to evaluate an AIML expression without printing or returning any result. The tag name is suggested by the expression “think but don’t speak”.

#### w. **<explode>**

The `<explode>` tag implements a string-manipulation function to split a word or sentence into individual letters.

`<explode>X</explode>` evaluates the contents of X, and returns a string consisting of each character of X separated by one space.

Example usage:

```
<category>
<pattern>EXPLODE *</pattern>
<template><explode><star/></explode>
</template>
</category>
```

Example dialog:

Human: Explode ABCDEF  
 Robot: A B C D E F  
 Human: Explode Dr. Alan Turing  
 Robot: D r A l a n T u r i n g

Note that the inverse operation, `implode`, is not defined in AIML because it’s easy to define the `implode` function with a couple AIML categories:

```
<category>
<pattern>IMPLode *</pattern>
<template><star/></template>
</category>
<category>
<pattern>IMPLode * *</pattern>
<template><star/><srai>IMPLode <star index="2"/></srai></template>
</category>
```

#### x. **<normalize> and <denormalize>**

The `<normalize>` tag processes the evaluated tag contents with the AIML normalization preprocessor.

The `<denormalize>` tag implements the approximate inverse of `<normalize>`.

The exact specification of the substitutions for `<normalize>` and `<denormalize>` are left up to the botmaster.

The following table shows some typical transformations.

x	<code>&lt;normalize&gt;x&lt;/normalize&gt;</code>	<code>&lt;denormalize&gt;</code>
---	---	----------------------------------

		<code>&lt;normalize&gt;x&lt;/normalize&gt; &lt;/denormalize&gt;</code>
callmom-info@pandorabots.com	callmom dash info at pandorabots dot com	callmom-info@pandorabots.com
(212)-333-4444	212 dash 333 dash 4444	212-333-4444
http://www.alicebot.org	http colon slash slash www.alicebot dot org	http://www.alicebot.org
I can not hear you	I can not hear you	I can't hear you

**y. <formal>**

The `<formal>` tag converts every word in the evaluated tag contents to title case, where the first letter of each word is capitalized and the remaining letters are in lower case.

**z. <lowercase>**

The `<lowercase>` tag converts the evaluated tag contents to all lower case.

**aa. <uppercase>**

The `<uppercase>` tag converts the evaluated tag contents to all upper case.

**bb. <sentence>**

The `<sentence>` tag converts the contents to sentence case, namely, the first letter of the first word is capitalized.

**cc. <person>**

The `<person>` tag applies a pronoun-swapping substitution to the evaluated tag contents. The exact method of specifying the person substitution is left up to the interpreter, and the exact substitutions themselves are left up to the botmaster.

The `<person>` tag was motivated by the ELIZA program, which used a similar pronoun-swapping trick to generate dialogs like:

**young woman:** You are not very aggressive but I think you don't want me to notice that.

**eliza:** What makes you think I am not very aggressive?

**young woman:** You don't argue with me.

**eliza:** Why do you think I don't argue with you?

**young woman:** You are afraid of me.

**eliza:** Does it please you to believe I am afraid of you?

(source: <http://www.stanford.edu/group/SHR/4-2/text/dialogues.html>)

Such a dialog can be created by AIML categories like:

```
<category>
<pattern>YOU ARE NOT * BUT *</pattern>
<template>What makes you think I am not <person><star/></person>?</template>
</category>
<category>
<pattern>YOU DO NOT *</pattern>
<template>Why do you think I do not <person><star/></person>?</template>
</category>
<category>
<pattern>YOU ARE *</pattern>
<template>Does it please you to believe I am <person><star/></person>?</template>
</category>
```

The `<person>` tag also exists in a singleton form, and may be defined as a macro:

```
<person/> = <person><star/></person>
```

**dd. <person2>**

`<person2>` is exactly like `<person>`, except that it is defined to swap 1st and 3rd person pronouns rather than 1st and 2nd.

The `<person2>` tag also exists in a singleton form, and may be defined as a macro:

```
<person2/> = <person2><star/></person2>
```

**ee. <star index="n"/>**

`<star index="n"/>` is replaced with the value of the sequence of input words matching the nth wildcard or AIML Set in the pattern.

`<star/>` is equivalent to `<star index="1"/>`.

For example if the pattern is `<pattern>_ I LIKE <set>color</set> *</pattern>` and the input is

“You know I like red carnations and roses”, then

```
<star/> = You know
```

```
<star index="2"/> = red
```

```
<star index="3"/> = carnations and roses
```

Like other AIML 2.0 tags, the attribute value of `<star/>` may appear in a subtag.

`<star index="3"/>` is equivalent to `<star><index>3</index></star>`.

#### ff. `<thatstar index="n"/>`

`<thatstar index="n"/>` is exactly like `<star/>`, except that it replaced with the value of the nth wildcard or set expression from the `<that>` pattern.

#### gg. `<topicstar index="n"/>`

`<thatstar index="n"/>` is exactly like `<star/>` and `<thatstar/>`, except that it replaced with the value of the nth wildcard or set expression from the `<topic>` pattern.

#### hh. `<that index="m,n"/>`

`<that/>` is replaced with the last sentence of the robot's last response.

`<that index="m,n"/>` refers to the nth last sentence of the mth last response.

`<that/>` is equivalent to `<that index="1,1"/>`

Like other AIML 2.0 tags, the attribute value of `<that/>` may appear in a subtag.

`<that index="2,3"/>` is equivalent to `<that><index>2,3</index></that>`

#### ii. `<input index="n"/>`

`<input index="n"/>` is replaced with the value of the nth previous sentence input to the bot.

Like other AIML 2.0 tags, the attribute value of `<input/>` may appear in a subtag.

`<input index="3"/>` is equivalent to `<input><index>3</index></input>`

#### jj. `<request index="n"/>`

`<request index="n"/>` is replaced with the value of the nth previous multi-sentence input to the bot.

Like other AIML 2.0 tags, the attribute value of `<request/>` may appear in a subtag.

`<request index="3"/>` is equivalent to `<request><index>3</index></request>`

#### kk. `<response index="n"/>`

`<response index="n"/>` is replaced with the value of the nth previous multi-sentence bot response..

Like other AIML 2.0 tags, the attribute value of `<response/>` may appear in a subtag.

`<response index="3"/>` is equivalent to `<response><index>3</index></response>`

## II. `<learn>`

The AIML 2.0 `<learn>` tag provides a way for the bot to learn new AIML categories dynamically through user interaction. In other words, the client can teach the bot new responses with `<learn>`. The categories learned with the `<learn>` tag are specific to the client chatting with the bot. In a multi-client system, if one client teaches the bot the response to “What is my sister’s name?”, then the response is specific to that client, and another client can teach the same bot a different response to the same input.

The `<learn>` and `<learnf>` tag make use of a companion tag, `<eval>`.

#### mm. `<learnf>`

`<learnf>` is exactly like `<learn>`, except that it saves the learned categories in an AIML file that is global to the bot. In a multi-client AIML system, if one client teaches a bot a category with `<learnf>`, the category may be activated by any other client chatting with the bot.

#### nn. `<eval>`

In a `<learn>` or `<learnf>` expression, the `<eval>` tag serves to insert the result of evaluating an AIML template expression.

#### oo. `<id/>`

The `<id/>` tag returns to customer or client ID of the current conversation

#### pp. `<vocabulary/>`

The `<vocabulary/>` tag computes a set of words consisting of:

1. All the words in the pattern paths found in the bot's Graphmaster.
2. All the words in the AIML Sets defined for the bot.

Example:

```
<category>
```

```
<pattern>HOW BIG IS YOUR VOCABULARY</pattern>
```

```
<template>I can recognize <vocabulary/> words.</template>
```

```
</category>
```

#### qq. `<program/>`

The `<program/>` tag returns a program name/version string indentifying the AIML interpreter.

Example AIML:

```
<category>
<pattern>WHAT VERSION ARE YOU</pattern>
<template>This is <program/>.</template>
</category>
```

**Sample dialog:**

Human: What version are you?

Robot: The is Program AB Version 0.0.2.0 beta

**rr. <system>**

The &lt;system&gt; tag executes a runtime execution under the operating system of the evaluated tag contents.

Example:

```
<category>
<pattern>LIST YOUR AIML FILES</pattern>
<template><system>dir bots\<bot name="name"/>\aiml</system>
</template>
</category>
```

**ss. <interval>**

The &lt;interval&gt; tag computes a time interval between two dates.

M

Like other AIML 2.0 tags, the attribute values of &lt;interval&gt; may be specified as XML attributes or with subtags of the same name. The attributes of &lt;interval&gt; are:

&lt;style&gt; - minutes, hours, days, weeks, months, or years

&lt;jformat&gt; - the format of the dates in the interval. (See &lt;date/&gt;).

&lt;from&gt; - starting date

&lt;to&gt; - ending date

Examples:

Compute the number of days until Christmas:

```
<category>
<pattern>HOW MANY DAYS UNTIL CHRISTMAS</pattern>
<template>
<interval>
<jformat>MMMMMMMM dd</jformat>
<style>days</style>
<from><date jformat="MMMMMMMM dd"/></from>
<to>December 25</to>
</interval>
days until Christmas.
</template>
</category>
```

The following category displays the bot's age in years, or in months if the bot is less than one year old.

```
<category><pattern>AGE</pattern>
<template>
<think>
<set var="years">
<interval>
<jformat>MMMMMMMM dd, YYYY</jformat>
<style>years</style>
<from>October 9, 2012</from>
<to><date jformat="MMMMMMMM dd, YYYY"/></to>
</interval>
</set>
<set var="months">
<interval>
<jformat>MMMMMMMM dd, YYYY</jformat>
<style>months</style>
<from>October 9, 2012</from>
<to><date jformat="MMMMMMMM dd, YYYY"/></to>
</interval>
```

```

</set>
</think>
<condition var="years">
<li value="0">I am <get var="months"/> months old.</li>
<li>I am <get var="years"/> years old.</li>
</condition>
</template>
</category>

```

**tt. <size/>**

The <size/> tag returns the number of AIML categories stored in the graph.

Example:

```

<category>
<pattern>HOW BIG ARE YOU</pattern>
<template>My brain contains <size/> categories.</template>
</category>

```

Human: How big are you?

Robot: My brain contains 7093 categories.

## 7. AIML Pattern Matching

Every AIML category is uniquely specified by an input pattern, that pattern and topic pattern. Remember, if the <that> and/or <topic> are left unspecified, they are assumed to have a default value of \*. A Pattern Path is defined as a linked sequence of nodes, where the nodes are linked by edges labeled with the words. The sequence of words in a Pattern Path is specified as the words from the input pattern, followed by the symbol <that>, the words in the that pattern, followed by the words in the topic pattern.

Figure 1 depicts the Pattern Path for a category

```

<category>
<pattern>DO YOU HAVE A DOG</pattern>
<template>Can your dog be my pet too?</template>
</category>

```

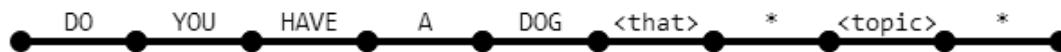


Figure 1. Pattern Path for a category

The AIML interpreter builds an object called the Graphmaster by reading the AIML files, constructing a Pattern Path for each category, and inserting the path into a directed, rooted graph. At the end of each path, the Graphmaster contains a link to the AIML template for the associated category.

Figure 2 shows an example of a simple Graphmaster for a bot with five categories.

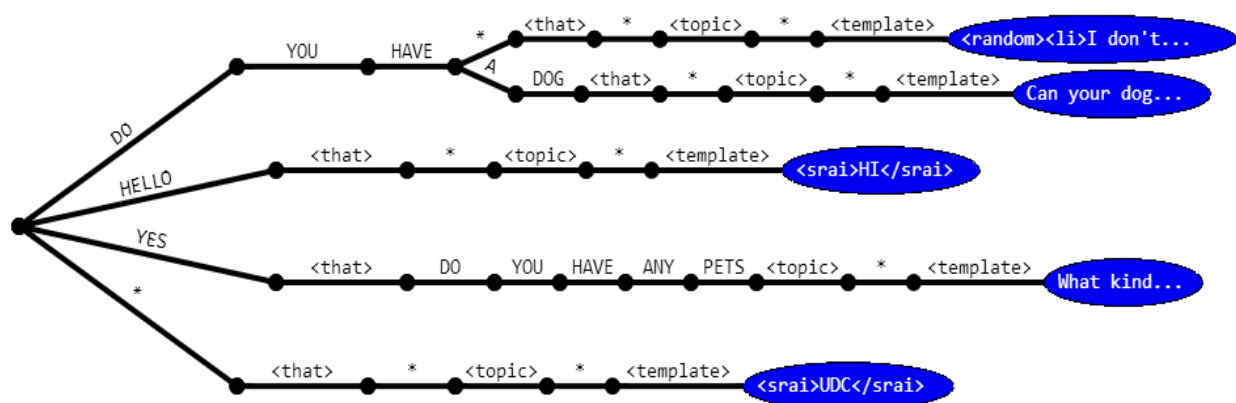


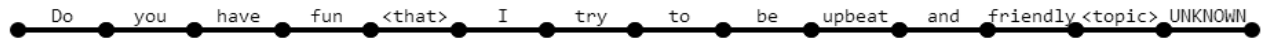
Figure 2. Simple Graphmaster with five categories

Given a specific input to the bot, the AIML interpreter builds an Input Path, similar to a Pattern Path, containing the normalized input, the bot's last reply (the value of `<that/>`)--also normalized, and the normalized topic. Figure 3 illustrates an example of an Input path resulting from the conversation fragment:

Robot: I try to be upbeat and friendly.

Human: Do you have fun?

For the purpose of this example, the topic is "unknown".

Figure 3. Input Path with `<that>` and `<topic>`

The AIML pattern matching algorithm searches the Graphmaster for a match of the Input Path. The search proceeds in a depth-first sequence. When searching a branch of the graph fails to find a match, the search algorithm backtracks to the last node with unexplored branches and searches those.

The search sequence at each node is guided by the following sequence:

1. `$word`
2. `#`
3. `_`
4. `word`
5. `<set>name</set>`
6. `^`
7. `*`

Or in plainer English,

1. dollar match - top priority word match
2. sharp match - zero+ word wildcard match
3. underscore match - one+ word wildcard match
4. word match - exact word match
5. set match - match found in AIML Set
6. caret match - zero+ wildcard match
7. star match - one+ word wildcard match

The wildcard `#` can match zero or more words from the input path.

The wildcard `_` can match one or more words from the input path.

For an exact word match, the next word in the input path must be identical (up to case invariance) with the word labeling the branch.

A set match also consumes one or more words, like a wildcard, but the word sequence must be a member of the named AIML set (see Sets and Maps in AIML 2.0).

The wildcard `^` can match zero or more words from the input path.

Finally, the wildcard `*` can match one or more words.

For the purpose of matching, the special symbols `<that>` and `<topic>` in the pattern and input paths are treated like exact words.

If no match at all is found in the graph, the interpreter should return a default string like "I have no answer for that", specified by the botmaster.

When matching wildcards and set items, the pattern matching algorithm contains appropriate bookkeeping functions to index and store the values of `<star/>`, `<thatstar/>` and `<topicstar/>`.

## Non-greedy

It is important to note that the AIML pattern matching is non-greedy. If the pattern contains a sequence of multiple wildcards, each wildcard except the last will consume one word of the input. For example, if the pattern is

`<pattern>* * *</pattern>`

and the matching input is "First second third fourth fifth", then the following should be true on the template side:

`<star/> = First`

```
<star index="2"/> = second
<star index="3"/> = third fourth fifth
```

## Graph implementation

If the graph nodes are implemented with hash tables, it is possible to find an exact word match in  $O(1)$  time. That is, provided the input path matches a path in the graph word-for-word, with no wildcards, the number of steps to find the response is proportional to the length of the path, and does not depend on the size of the graph.

Pathological cases exist however such as a long sequence of wildcards like:

```
<pattern>*****</pattern>
```

In a simple backtracking implementation, the matcher might try to match an input of 15 words with a 16 wildcard pattern. The algorithm would work its way through a vast number of combinations, trying one word for each wildcard, then two words, three and so on, but never finding a match with this pattern. To mitigate this problem, the graph nodes can have a height property defined as the minimum number of words in the input path needed to reach a leaf from that node.

## Duplicate Categories

Two categories are said to be *duplicates* if they have the same pattern, that and topic. A pair of duplicate categories may have different templates. As the AIML interpreter loads categories from AIML files, it may encounter duplicates. The method for handling duplicates is called the merge policy. The interpreter should give the botmaster some control over the merge policy.

Some typical merge policies are:

1. Use first loaded - Keep the first duplicate loaded, and discard any subsequent ones.
2. Use last loaded - Keep the last duplicate category loaded, and discard any previous ones.
3. Merge with <random> - If the two duplicate categories' templates differ, combine them by placing each in a <random> list item.

### 8. AIML Intermediate Format (or "I'm flattened")

An AIML bot typically consists of thousands, or hundreds of thousands, of AIML categories. Each category contains an input pattern and a template, and optionally a <that> pattern and a <topic> pattern. When we think of AIML content this way, it brings to mind row-oriented data that might be found in a database or spreadsheet. You can imagine AIML represented in a table, where the rows are the categories and the columns are labeled "pattern", "that", "topic" and "template". What's more, there are other attributes that can be associated with each category, for example the number of times each category is activated, and the name of the AIML source file containing the category. The imaginary spreadsheet might also include columns called "activation count" and "filename".

Yet the AIML template is more like a hierarchical, tree-like structure. The simplest AIML template contains only text, but AIML allows the text to be marked up with AIML tags. These tags might enclose more text and more tags, giving rise to a structure like a tree. You can think of the template as the root of a tree, with branches leading to tags and text.

AIML is therefore a hybrid of database-style row data, and the hierarchical data in the templates. Because AIML is based on XML, and there is a significant amount of software written in a variety of languages for parsing XML, many AIML interpreters simply read the AIML files and process them as hierarchical data. An XML parser reads the AIML file, and parses it into a collection of nodes representing categories, and the nodes are further parsed into the component pattern, that, topic and template parts. The template is decomposed into its constituent hierarchical structure.

The problem with this approach is that, because of the size of the AIML files, the full XML parsing can be relatively slow. Also, there is no real need to parse all the individual templates until they are activated. For these reasons, we defined an intermediate AIML format that represents the categories as row data, and stores the templates as XML data. This format is called AIML Intermediate Format, or AIMLIF.

AIMLIF is a plain text format representing categories as line data.

One category per line, one line per category:

```
ACTIVATION_COUNT,PATTERN,THAT, TOPIC, TEMPLATE,FILENAME
```

In TEMPLATE, we replace "\n" with "#\Newline" and "," with "#\Comma", so that each category takes only one line of text.

Let's look at an example. This AIML category comes from the file reductions.aiml:

```
<category>
  <pattern>DO YOU THINK YOU WILL *</pattern>
  <template><srai>WILL YOU <star/></srai>
</template>
</category>
```

has an intermediate format representation

```
0,DO YOU THINK YOU WILL *,*,*,<srai>WILL YOU <star/></srai>,reductions.aiml
```

By preprocessing the AIML files into AIMLIF, our program can load the AIML much faster. Instead of parsing all the AIML XML files at load time, the program loads the AIMLIF files and stores the <template> XML for later use. The <template> XML is parsed only when the category is activated.

### Unix Tools

A pleasant side-effect of using AIMLIF representation is that it facilitates applying common Unix tools to analyze and process the AIML. Because each category is represented as a single line of text, we can use tools like sed, awk, grep, sort and uniq on the AIMLIF files easily.

### CSV File Format

The AIMLIF format is easily recognizable as the familiar spreadsheet [.csv file format](#). AIMLIF files can be read and edited with spreadsheet tools, including MS Excel and Google Docs, facilitating such features as upload/download of AIML files in .csv format, and the development of customized spreadsheet editors for AIML.

#### Changing CSV delimiter

The “,” (comma) character is slightly problematic as a field separator for AIMLIF because the comma is common in <template> expressions. If you are editing your AIML files in CSV format using Excel, take care not to use “,” in the <template>. A stray comma can cause the AIMLIF expression

```
0,HELLO,*,*,Hi, friend,personality.aiml
```

to be interpreted as a category

```
<category>
<pattern>HELLO</pattern>
<template>Hi</template>
</category>
```

saved in a file called “friend” instead of “personality.aiml”.

The recommended solution is to use the symbol #Comma, as in

```
0,HELLO,*,*,Hi#Comma friend,personality.aiml
```

If you are using Windows however, it's possible to change the default delimiter under Windows under Control Panel-->Region and Language-->Additional Settings. A good choice might be “.” or “|” because these are less common in <template> expressions. If you are using Program AB to convert between CSV and AIML files, you need to set the Magic String aimlif\_split\_char to your new delimiter.

For a discussion about changing the default list separator in Windows, see <http://www.techmickey.org/how-to-change-delimiters-in-excel-to-open-csv-file-using-semi-colon-and-comma-as-delimiters-in-csv-files/>

Note: we need add a configuration option in Program AB to make it possible to specify a different list separator.

### Appendix I. OOB Tags

AIML 2.0 includes a new set of tags to control device actions called OOB tags. "OOB" means "Out of Band", an engineering term for a conversation on a separate, hidden channel. For example if you are having a phone call with someone, and during the call send them a text message, the text message is "out of band" from the voice call. In our case this refers to commands that the bot sends to the phone as part of a reply, but these commands are hidden from the end-user.

AIML 2.0 includes the OOB tag specification but it important to note that OOB tag processing adds a second pass of XML processing to an AIML 2.0 interpreter. In the first pass, the interpreter processes an input, evaluates any template tags, and produces a result. This result may contain unevaluated OOB tags. The OOB tags are processed in a second phase.

The two-phase model achieves a clean separation between “AI functions” and “device functions”. The AIML interpreter can be written as a library, or even run on a remote server, and a more lightweight app running on a device can process the OOB commands.

The two-phase model also allows the AIML to dynamically write the contents of the OOB tags.

The complete description of OOB tags may be found in a companion document:

<http://code.google.com/p/aiml-en-us-pandorabots-callmom/wiki/CallMomOOBTags>



## References

1. [AIML OOB Tags](#)
2. [Sets and Maps in AIML 2.0](#)
3. [Artificial Intelligence Markup Language \(AIML\) Version 1.0.1](#)

## Glossary

**AIML File** - an XML format file containing AIML categories

**AIML Interpreter** - a program that can load and run an AIML bot and provide responses to conversational requests according to the AIML specification in this document.

**AIML Map** - a function that computes a member of one AIML Set from another.

**AIML Set** - a collection of strings (words and phrases) that can be matched in an input.

**Bot** - a collection of AIML files, configuration files, and AIML Sets and Maps, serving conversational requests in an AIML interpreter.

**Bot property** - a global constant value for a bot.

**Botmaster** - the author of an AIML bot.

**Category** - The basic unit of knowledge in AIML, consisting of an input pattern, response template and optionally a that pattern and topic pattern.

**Client** - a person (or other program) chatting with a bot.

**Default category** - a category with a pattern containing a wildcard.

**Depth-first Search** - The method of searching the Graphmaster for a match (see [http://en.wikipedia.org/wiki/Depth-first\\_search](http://en.wikipedia.org/wiki/Depth-first_search)).

**Duplicate categories** - a pair of categories with the same input pattern, that pattern and topic pattern (but not necessarily the same template).

**Graphmaster** - the object storing the AIML categories in a tree, where each category is uniquely identified by a path from the root to a leaf node.

**Input** - A single sentence transmitted to the bot.

**Input path** - A sequence formed by combining an input sentence, the robot's last reply ("that") and the topic.

**Knowledge Base** - another name for the bot's AIML files.

**Map** - see AIML Map

**Normalization** - a process that applies a series of substitutions to an input to put it into a standard format for AIML pattern matching. Typically normalization removes punctuation, corrects some spelling mistakes, and expands contractions.

**One+ wildcard** - a wildcard that can match one or more words.

**Out-of-Band (OOB)** - XML embedded in the AIML response that is not interpreted by the AIML Interpreter, but passed through to a secondary process that takes some action based on the OOB command.

**Pattern Path** - a sequence formed by combining an input pattern, that pattern and topic pattern.

**Predicate** - a variable specific to a client.

**Recursion** - another name for reduction.

**Reduction** - An operation using the AIML <srai> tag that simplifies, translates, rewrites, or reduces the input into another form, and then sends that form back to the Graphmaster to match another AIML category.

**Set** - see AIML set.

**Symbolic reduction** - another name for reduction.

**Tag** - an XML symbol denoting the beginning and end of an XML expression.

**Template** - the response part of an AIML category. The template consists of text and AIML markup.

**That** - The last sentence of the robot's last reply.

**Topic** - A global state variable that may be set in a template, and used to control category matching.

**Ultimate Default Category** - The AIML category containing a pattern with the wildcard \* by itself, meaning that this category matches no words in the input. The UDC is the category of last resort.

**Wildcard** - a symbol in an AIML pattern expression can match any words in the input.

**XML** - <http://en.wikipedia.org/wiki/XML>

**Zero+ Wildcard** - a wildcard that can match zero or more words.

---

公開元: [Google ドライブ](#) – [不正行為の報告](#) – 5 分ごとに自動更新

---