Artificial Intelligence Markup Language (AIML) Version 1.0.1

A.L.I.C.E. AI Foundation
Official AIML 1.0.1 standard
Adopted October 30, 2011 (rev 008)

This document resides at:

http://alicebot.org/TR/2011/WD-aiml/index.html

Working Draft versions:

http://alicebot.org/TR/2005/WD-aiml-1.0.1-008.html

http://alicebot.org/TR/2001/WD-aiml-1.0.1-20010218-007.html

http://alicebot.org/TR/2001/WD-aiml-1.0.1-20011018-005.html

http://alicebot.org/TR/2001/WD-aiml-1.0-20010926-004.html

http://alicebot.org/TR/2001/WD-aiml-1.0-20010926-003.html

Creator of AIML:

Richard Wallace

Editor of this Document:

Richard Wallace

Original Author of this Document(*):

Noel Bush

Contributors to source material for this document:

Thomas Ringate, Anthony Taylor, Jon Baer, Noel Bush(*)

(*) - Note: As of 2/2005, most of the material in this document was written by former A. I. Foundation member Noel Bush. His outstanding contribution to this documentation and standardization effort is greatly appreciated by the A. I. Foundation and by members of the A.L.I.C.E. AI community.

Abstract

The Artificial Intelligence Markup Language is a derivative of XML (Extensible Markup Language) that is completely described in this document. Its goal is to enable pattern-based, stimulus-response knowledge content to be served, received and processed on the Web and offline in the manner that is presently possible with HTML and XML. AIML has been designed for ease of implementation, ease of use by newcomers, and for interoperability with XML and XML derivatives such as XHTML.

Status of this Document

This document was drafted for review by the AIML Architecture Committee A.L.I.C.E. AI Foundation, with the intention of providing a more thorough specification than that laid out in the "AIML 1.0 Tag Set". It was released by the Committee as a Working Draft to gather public feedback before its promotion to a Proposal, and eventually to this final release as the AIML 1.0.1 Standard.

This is not a signficantly new version of AIML (first published 3 August 2001 at http://alicebot.org/committees/architecture/resolutions/aiml10.html); rather, it incorporates small amendments made by the Architecture Committee during the period of August 2001-September 2011 and provides a more formal statement of the specification than has been available thus far. According to the Committee's current specification release process, the incorporation of these changes requires that the specification be considered a new specification; the addition of the subminor ".1" to "1.0" reflects this.

Please report errors in this document to drwallace@alicebot.org.

Table of Contents

1. Introduction

Artificial Intelligence Markup Language, abbreviated AIML, describes a class of data objects called AIML objects and partially describes the behavior of computer programs that process them. AIML is a derivative of XML, the Extensible Markup Language. By construction, AIML objects are conforming XML documents, although AIML objects may also be contained within XML documents. As XML is itself an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879], AIML objects are also conforming SGML documents.

AIML objects are made up of units called topics and categories, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form AIML elements. AIML elements encapsulate the stimulus-response knowledge contained in the document. Character data within these elements is sometimes parsed by an AIML interpreter, and sometimes left unparsed for later processing by a Responder.

[Definition: A software module called an **AIML interpreter** is used to read AIML objects and provide application-level functionality based on their structure. An AIML interpreter may use the services of an XML processor, or it may take the place of one, but it must not violate any of the constraints defined for XML processors.] [Definition: It is assumed that an AIML interpreter is part of a larger application generically termed a **bot**, which carries the larger functional set of interaction based on AIML. This document does not constrain the specific behavior of a bot.] [Definition: A software module called a **responder** handles the human-to-bot or bot-to-bot interface work between an AIML interpreter and its object(s).]

1.1. Origin and Goals

AIML was developed by Dr. Richard S. Wallace and the Alicebot free software community during 1995-2000. It was originally adapted from a non-XML grammar also called AIML, and formed the basis for the first Alicebot, A.L.I.C.E., the Artificial Linguistic Internet Computer Entity. Since its inception, it has been adopted as a standard by the A.L.I.C.E. AI Foundation, which now holds its copyright, and whose Alicebot and AIML Architecture Committee is responsible for its maintenance and further elaboration.

The design goals for AIML are:

1. AIML shall be easy for people to learn.
2. AIML shall encode the minimal concept set necessary to enable a stimulus-response knowledge system modeled on that of the original A.L.I.C.E.
3. AIML shall be compatible with XML.
4. It shall be easy to write programs that process AIML documents.
5. AIML objects should be human-legible and reasonably clear.
6. The design of AIML shall be formal and concise.
7. AIML shall not incorporate dependencies upon any other language.

This specification provides all the information necessary to understand AIML Version 1.0.1 and construct computer programs to process it.

This version of the AIML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2. Terminology

Like much of this document, the terminology used in this document to describe AIML objects is borrowed freely from the W3C's XML Recommendation (http://www.w3.org/TR/REC-xml). The following terms, copied almost verbatim from the XML Recommendation, are used in building those definitions and in describing the actions of an AIML interpreter:

may

[Definition: Conforming objects and AIML interpreters are permitted to but need not behave as described.]

must

[Definition: Conforming objects and AIML interpreters are required to behave as described; otherwise they are in error.]

error

[Definition: A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.]

fatal error

[Definition: An error which a conforming AIML interpreter must detect and report to the bot. After encountering a fatal error, the interpreter may continue processing the data to search for further errors and may report such errors to the bot. In order to support correction of errors, the interpreter may make unprocessed data from the object (with intermingled character data and AIML content) available to the bot. Once a fatal error is detected, however, the interpreter must not continue normal processing (i.e., it must not continue to pass character data and information about the object's logical structure to the bot in the normal

way).]

at user option

[Definition: Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.]

validity constraint

[Definition: A rule that applies to all valid AIML objects. Violations of validity constraints are errors; they must, at user option, be reported by validating AIML interpreters.

well-formedness constraint

[Definition: A rule that applies to all well-formed XML documents, and hence to all well-formed AIML objects. Violations of well-formedness constraints are fatal errors, as in the XML specification.]

for compatibility

[Definition: Marks a sentence describing a feature of AIML included solely to ensure that AIML remains compatible with XML.]

2. AIML Objects

[Definition: A data object is an **AIML object** if it is well-formed, as defined in the XML specification, and if it is valid according to the AIML specification--that is, if it conforms to the data model described in this document.]

Each AIML object has both a logical and a physical structure. Physically, the object is composed of units called topics and categories. An object begins in a "root" or object entity. Logically, the object is composed of elements and character references, all of which are indicated in the object by explicit markup.

An AIML object may also be "overlaid" by comments and processing instructions as described by the XML specification, as well as by XML content from other namespaces. Comments and processing instructions are not treated by an AIML interpreter. Foreign-namespace content may be passed by an AIML interpreter to a responder, but is not processed by the AIML interpreter itself.

In the following, we reiterate the AIML definitions of various aspects of object structure, although in the great majority of cases these simply repeat the XML definitions of the same. In several cases, there are further constraints on AIML interpreters that do not apply to all XML processors.

2.1. Well-formed AIML Objects

[Definition: A textual object is a **well-formed AIML object** if it meets the criteria specified in this document and matches the definition of XML well-formedness (http://www.w3.org/TR/2000/REC-xml-20001006#sec-well-formed).]

2.2. Characters

The AIML definitions for **text** and **characters** are adopted verbatim from the XML specification:

http://www.w3.org/TR/2000/REC-xml-20001006#charsets.

2.3. Common Syntactic Constructs

This document will use the same symbols used in the XML specification: http://www.w3.org/TR/2000/REC-xml-20001006#sec-common-syn.

2.4. Character Data and Markup

AIML uses the same definitions of **character data** and **markup** as used in the XML specification: http://www.w3.org/TR/2000/REC-xml-20001006#syntax.

2.5. Comments

AIML respects the same definition of **comments** as in XML: http://www.w3.org/TR/2000/REC-xml-20001006#sec-comments.

However, an AIML interpreter must *not* make it possible for a bot to retrieve the text of comments. Comments are regarded as a "wholly XML" feature, not at all part of the functional set of AIML interpretation.

2.6. Processing Instructions

AIML respects the same definition of **processing instructions** as in XML: http://www.w3.org/TR/2000/REC-xml-20001006#sec-pi. AIML does not make special use of the processing instruction function.

2.7. CDATA Sections

As in XML (http://www.w3.org/TR/2000/REC-xml-20001006#sec-cdata-sect), **CDATA sections** may occur anywhere character data may occur in AIML, and are used to escape blocks of text containing characters that would otherwise be recognized as markup.

However, CDATA sections must not be used as a substitute for proper namespace qualification; that is, CDATA sections must not contain markup from other namespaces that is meant to be interpreted later by a responder.

2.8. Prolog and Document Type Declaration

AIML adds no further constraints on the XML specification of prologs (http://www.w3.org/TR/2000/REC-xml-20001006#sec-prolog-dtd).

Since AIML objects can appear within other XML documents, which themselves may or may not have prologs, AIML interpreters cannot require a prolog to be present.

AIML does not use Document Type Declarations. Instead, AIML uses W3C Schemas. This document may be understood as the description of the AIML schema. AIML objects are not required to refer to an AIML schema namespace; however, this is strongly advised since the AIML specification is subject to change and namespace identification will aid AIML interpreters in deciding upon proper interpretation of AIML objects.

2.9. Standalone Document Declaration

AIML shares the XML definition of an **external markup declaration** (http://www.w3.org/TR/2000/REC-xml-20001006#sec-rmd). AIML interpreters should make use of such declarations in interpreting AIML objects, and should pass along appropriate declaration information to responders.

2.10. White Space Handling

AIML interpreters must respect the entirety of the XML definition of whitespace handling (http://www.w3.org/TR/2000/REC-xml-20001006#sec-white-space), including the use of the special attribute xml:space.

The default behavior of AIML interpreters when handling whitespace must be:

1. in cases where elements abut character data or other elements, with no intervening whitespace, to preserve this arrangement in the output;
2. in cases where at least one whitespace character in the AIML object intervenes between an element and character data or another element, to produce exactly one space character regardless of how many whitespace characters are present in the document, *unless* the xml:space attribute is applied to the parent element;
3. in cases where the xml:space is applied to an element, to respect the precise whitespace content of the element.

2.11. End-of-Line Handling

AIML interpreters should respect the end-of-line handling requirements of the XML specification (http://www.w3.org/TR/2000/REC-xml-20001006#sec-line-ends).

2.12. Language Identification

AIML objects may make use of the XML attribute xml:lang for identifying the natural or formal language in which the content of an object is written. (See http://www.w3.org/TR/2000/REC-xml-20001006#sec-lang-tag).

AIML is meant to be independent of human language, and thus the optional XML method of providing such information is the only means available to AIML authors; no additional entities support this kind of explicit markup.

3. AIML Object Structure

3.1. AIML Namespace

The AIML 1.0.1 namespace (herein referred to as "the AIML namespace") has the URIhttp://alicebot.org/2001/AIML-1.0.1.

AIML interpreters must use the XML namespaces mechanism [XML Names] to recognize elements and attributes from this namespace. The complete list of AIML-defined elements is defined in this document.

Vendors must not extend the AIML namespace with additional elements or attributes. Instead, any extension must be in a separate namespace.

This specification uses a prefix of aiml: for referring to elements in the AIML namespace. However, AIML objects are free to use any prefix, provided that there is a namespace declaration that binds the prefix to the URI of the AIML namespace.

An element from the AIML namespace may have any attribute not from the AIML namespace, provided that the expanded-name of the attribute has a non-null namespace URI. The presence of such attributes must not change the behavior of AIML units and functions described in this document. Thus, an AIML interpreter is always free to ignore such attributes, and must ignore such attributes without giving an error if it does not recognize the namespace URI. Such attributes can provide, for example, unique identifiers, optimization hints, or documentation.

It is an error for an element from the AIML namespace to have attributes with expanded-names that have null namespace URIs (i.e., attributes with unprefixed names) other than attributes defined for the element in this document.

NOTE: The conventions used for the names of AIML elements and attributes are that names are all lower-case, use hyphens to separate words, and use abbreviations only in support of historical usage (as in <li>).

3.2. AIML Element

<aiml:aiml

  version = *number*>

  <!-Content: *top-level-elements*) -->

</aiml:aiml>

An AIML object is represented by an aiml:aiml element in an XML document.

An AIML object must have a version attribute, indicating the version of AIML that the object requires. For this version of AIML, the version should be 1.0.1. When the value is not equal to 1.0.1, forward-compatible processing mode is enabled.

The aiml:aiml element may contain the following types of elements:

- aiml:topic
- aiml:category

An element occurring as a child of an aiml:aiml element is called a **top-level** element.

This example shows the top-level structure of an AIML object. Ellipses (...) indicate where attribute values or content have been omitted. AIML objects may contain zero or more of each of these elements.

<aiml:aiml version="1.0.1"
        xmlns:aiml="http://alicebot.org/2001/AIML">

  <aiml:topic name="...">

```
    ...
  </aiml:topic>
  <aiml:category>
      ...
  </aiml:category>
</aiml:aiml>
```

That is, an AIML object must contain zero or more topic elements, and one or more category elements.

The order in which the children of the aiml:aiml element occur is unconstrained (at user option). Users are free to order the elements as they prefer.

In addition, the aiml:aiml element may contain any element not from the AIML namespace, provided that the expanded-name of the element has a non-null namespace URI. The presence of such top-level elements must not change the behavior of AIML elements defined in this document. An AIML interpreter must always skip interpretation of such top-level elements, but must also pass these elements along to the responder. Such elements might be, for example, formatting elements from a namespace such as XHTML, SMIL, or XSL-FO.

3.3. Forward-Compatible Processing

An element enables forward-compatible mode for itself, its attributes, its descendants and their attributes if it is an aiml:aiml element whose version attribute is not equal to 1.0.1.

If an element is processed in forward-compatible mode, then:

- if it is a top-level element and AIML 1.0.1 does not allow such elements as top-level elements, then the element must be ignored along with its content;
- if it is an element in a category and AIML 1.0.1 does not allow such elements to occur in categories, then the element should be ignored;
- if the element has an attribute that AIML 1.0.1 does not allow the element to have or if the element has an optional attribute with a value that the AIML 1.0.1 does not allow the attribute to have, then the attribute must be ignored.

Thus, any AIML 1.0.1 interpreter must be able to process the following AIML object without error (but also without producing any behavior in response to the non-1.0.1 element):

```
<aiml:aiml  version="1.1"
        xmlns:aiml="http://alicebot.org/2001/AIML">
  <aiml:topic name="demonstration *">
    <aiml:category>
      <aiml:pattern>* EXAMPLE</aiml:pattern>
      <aiml:template>
        This is just an example, <get name="username"/>.
```

```
        <aiml:exciting-new-1.1-feature/>
      </aiml:template>
    </aiml:category>
  </aiml:topic>
```

3.4. AIML Pattern Expressions

Several AIML elements and element attributes require a restricted form of mixed character data and optional restricted element content that is called an **AIML Pattern Expression**. There are two types of pattern expressions: **simple pattern expressions** and **mixed pattern expressions**. Mixed pattern expressions cannot be fully described by either Document Type Declaration syntax or W3C Schema 1.0.1 syntax. The structure of AIML pattern expressions is given in [8.1].

3.5. AIML Predicate Handling

Many AIML elements attributes deal with **predicates**. The definition of an AIML predicate is found in [9], and the restrictions on predicate name are given in [9.1].

A special class of AIML predicates are **bot predicates**, which have the same name restrictions as all AIML predicates and whose values can only be retrieved, not set, at runtime. The values of bot predicates are set at load time.

3.6. Embedding AIML

An AIML object may be a standalone XML document with the aiml:aiml element as the document element, or it may be embedded in another resource. Two forms of embedding are possible:

- the AIML object may be textually embedded in a non-AIML resource, or
- the aiml:aiml element may occur in an XML document other than as the document element

AIML 1.0.1 does *not* currently permit an id attribute on the aiml:aiml element to facilitate the second form of embedding, such as XSLT provides (see http://www.w3.org/TR/xslt#section-Embedding-Stylesheets).

The following example shows how an XHTML document can contain an AIML object, such that an AIML interpreter could read the AIML object (including the "overlaid" XHTML content with which it coincides):

```
<!DOCTYPE
  html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=http://www.w3.org/1999/xhtml xmlns:aiml=http://alicebot.org/2001/AIML xml:lang="en-US">

  <head>

    <title>Our Company</title>

  </head>
```

```
<body>
  <aiml:aiml>
    <aiml:category>
      <aiml:pattern>
        <h1>About Our Company</h1>
      </aiml:pattern>
      <aiml:template>
        <p>
          We sell genetically engineered worms that clean the sidewalk and communicate via microwave transmissions.
        </p>
      </aiml:template>
    </aiml:category>
  </aiml:aiml>
</body>
</html>
```

(Note that indentation in this example is provided purely for clarity, but white space should be handled as described in [2.10.].)

4. Topic

A topic is an optional top-level element that contains category elements. A topic element has a required name attribute that must contain a simple pattern expression. A topic element may contain one or more category elements.

The contents of the topic element's name attribute are appended to the full match path that is constructed by the AIML interpreter at load time, as described in [8.2].

```
<!-- Category: top-level-element -->

<aiml:topic
  name = aiml-simple-pattern-expression >

  <!-- Content: aiml:category+ -->

</aiml:topic>
```

5. Category

A category is a top-level (or second-level, if contained within a topic) element that contains exactly one pattern and exactly one template. A category does not have any attributes.

All category elements that do not occur as children of an explicit topic element must be assumed by the AIML interpreter to occur as children of an "implied" topic whose name attribute has the value * (single asterisk wildcard).

<!-- Category: top-level-element -->

<aiml:category>

  <!-- Content: *aiml-category-elements* -->

</aiml:category>

6. Pattern

A pattern is an element whose content is a mixed pattern expression. Exactly one pattern must appear in each category. The pattern must always be the first child element of the category. A pattern does not have any attributes.

The contents of the pattern are appended to the full match path that is constructed by the AIML interpreter at load time, as described in [8.2].

<!-- Category: aiml-category-elements -->

<aiml:pattern>

  <!-- Content: *aiml-pattern-expression* -->

</aiml:pattern>

6.1. Pattern-side That

The pattern-side that element is a special type of pattern element used for context matching. The pattern-side that is optional in a category, but if it occurs it must occur no more than once, and must immediately follow the pattern and immediately precede the template. A pattern-side that element contains a simple pattern expression.

The contents of the pattern-side that are appended to the full match path that is constructed by the AIML interpreter at load time, as described in [8.2].

If a category does not contain a pattern-side that, the AIML interpreter must assume an "implied" pattern-side that containing the pattern expression * (single asterisk wildcard).

A pattern-side that element has no attributes.

<!-- Category: aiml-category-elements -->

```
<aiml:that>
  <!-- Content: aiml-pattern-expression -->
</aiml:that>
```

7. Template

A template is an element that appears within category elements. The template must follow the pattern-side that element, if it exists; otherwise, it follows the pattern element. A template does not have any attributes.

```
<!-- Category: aiml-category-elements -->
<aiml:template>
  <!-- Content: aiml-template-elements -->
</aiml:template>
```

The majority of AIML content is within the template. The template may contain zero or more AIML template elements mixed with character data. The elements described below are grouped for convenience.

7.1. Atomic template elements

An atomic template element in AIML indicates to an AIML interpreter that it must return a value according to the functional meaning of the element. Atomic elements do not have any content.

7.1.1. Star

The star element indicates that an AIML interpreter should substitute the value "captured" by a particular wildcard from the pattern-specified portion of the match path when returning the template.

The star element has an optional integer index attribute that indicates which wildcard to use. The minimum acceptable value for the index is "1" (the first wildcard), and the maximum acceptable value is equal to the number of wildcards in the pattern.

An AIML interpreter should raise an error if the index attribute of a star specifies a wildcard that does not exist in the category element's pattern. Not specifying the index is the same as specifying an index of "1".

The star element does not have any content.

```
<!-- Category: aiml-template-elements -->
<aiml:star
  index = single-integer-index />
```

7.1.2. Template-side That

The template-side that element indicates that an AIML interpreter should substitute the contents of a previous bot output.

The template-side that has an optional index attribute that may contain either a single integer or a comma-separated pair of integers. The minimum value for either of the integers in the index is "1". The index tells the AIML interpreter *which* previous bot output should be returned (first dimension), and optionally which "sentence" (see [8.3.2.]) of the previous bot output (second dimension).

The AIML interpreter should raise an error if either of the specified index dimensions is invalid at run-time.

An unspecified index is the equivalent of "1,1". An unspecified second dimension of the index is the equivalent of specifying a "1" for the second dimension.

The template-side that element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:that
  index = (*single-integer-index* | *comma-separated-integer-pair*) />

7.1.3. Input

The input element tells the AIML interpreter that it should substitute the contents of a previous user input.

The template-side input has an optional index attribute that may contain either a single integer or a comma-separated pair of integers. The minimum value for either of the integers in the index is "1". The index tells the AIML interpreter *which* previous user input should be returned (first dimension), and optionally which "sentence" (see [8.3.2.]) of the previous user input.

The AIML interpreter should raise an error if either of the specified index dimensions is invalid at run-time.

An unspecified index is the equivalent of "1,1". An unspecified second dimension of the index is the equivalent of specifying a "1" for the second dimension.

The input element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:input
  index = (*single-integer-index* | *comma-separated-integer-pair*) />

7.1.4. Thatstar

The thatstar element tells the AIML interpreter that it should substitute the contents of a wildcard from a pattern-side that element.

The thatstar element has an optional integer index attribute that indicates which wildcard to use; the minimum acceptable value for the index is "1" (the first wildcard).

An AIML interpreter should raise an error if the index attribute of a star specifies a wildcard that does not exist in the that element's pattern content. Not specifying the index is the same as specifying an index of "1".

The thatstar element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:thatstar
index = *single-integer-index* />

7.1.5. Topicstar

The topicstar element tells the AIML interpreter that it should substitute the contents of a wildcard from the current topic (if the topic contains any wildcards).

The topicstar element has an optional integer index attribute that indicates which wildcard to use; the minimum acceptable value for the index is "1" (the first wildcard). Not specifying the index is the same as specifying an index of "1".

The topicstar element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:topicstar
index = *single-integer-index* />

7.1.6. Get

The get element tells the AIML interpreter that it should substitute the contents of a predicate, if that predicate has a value defined. If the predicate has no value defined, the AIML interpreter should substitute the empty string "".

The AIML interpreter implementation may optionally provide a mechanism that allows the AIML author to designate default values for certain predicates (see [9.3.]).

The get element must not perform any text formatting or other "normalization" on the predicate contents when returning them.

The get element has a required name attribute that identifies the predicate with an AIML predicate name.

The get element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:get
name = *aiml-predicate-name* />

7.1.6.1. Bot

An element called bot, which may be considered a restricted version of get, is used to tell the AIML interpreter that it should substitute the contents of a "bot predicate". The value of a bot predicate is set at load-time, and cannot be changed at run-time. The AIML interpreter may decide how to set the values of bot predicate at load-time. If the bot predicate has no value defined, the AIML interpreter should substitute an empty string.

The bot element has a required name attribute that identifies the bot predicate.

The bot element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:bot
  name = *aiml-predicate-name* />

7.1.7. Short-cut elements

Several atomic AIML elements are "short-cuts" for combinations of other AIML elements. They are listed here without further explanation; the reader should refer to the descriptions of the "long form" of each element for which the following elements are short-cuts.

7.1.7.1. SR

The sr element is a shortcut for:

<srai><star/></srai>

The atomic sr does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:sr/>

7.1.7.2. Person2

The atomic version of the person2 element is a shortcut for:

<person2><star/></person2>

The atomic person2 does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:person2/>

7.1.7.3. Person

The atomic version of the person element is a shortcut for:

<person><star/></person>

The atomic person does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:person/>

7.1.7.4. Gender

The atomic version of the gender element is a shortcut for:

<gender><star/></gender>

The atomic gender does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:gender/>

NB: Previous versions of AIML (dubbed "0.9") used <aiml:gender/> to indicate that the AIML interpreter should return the value of the bot predicate "gender". Vendors who wish to implement an AIML interpreter that is compatible with old AIML sets written with this usage should take note.

7.1.8. System-defined predicates

Several atomic AIML elements require the AIML interpreter to substitute a value that is determined from the system, independently of the AIML content.

7.1.8.1. Date

The date element tells the AIML interpreter that it should substitute the system local date and time. No formatting constraints on the output are specified.

The date element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:date/>

7.1.8.2. ID

The id element tells the AIML interpreter that it should substitute the user ID. The determination of the user ID is not specified, since it will vary by application. A suggested default return value is "localhost".

The id element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:id/>

7.1.8.3. Size

The size element tells the AIML interpreter that it should substitute the number of categories currently loaded.

The size element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:size/>

7.1.8.4. Version

The version element tells the AIML interpreter that it should substitute the version number of the AIML

interpreter.

The version element does not have any content.

<!-- Category: aiml-template-elements -->

<aiml:version/>

7.2. Text formatting elements

AIML contains several text-formatting elements, which instruct an AIML interpreter to perform locale-specific post-processing of the textual results of the processing of their contents.

7.2.1. Uppercase

The uppercase element tells the AIML interpreter to render the contents of the element in uppercase, as defined (if defined) by the locale indicated by the specified language (if specified).

<!-- Category: aiml-template-elements -->

<aiml:uppercase>

  <!-- Contents: *aiml-template-elements* -->

</aiml:uppercase>

If no character in this string has a different uppercase version, based on the Unicode standard, then the original string is returned.

See Unicode Case Mapping for implementation suggestions.

7.2.2. Lowercase

The lowercase element tells the AIML interpreter to render the contents of the element in lowercase, as defined (if defined) by the locale indicated by the specified language (if specified).

<!-- Category: aiml-template-elements -->

<aiml:lowercase>

  <!-- Content: *aiml-template-elements* -->

</aiml:lowercase>

If no character in this string has a different lowercase version, based on the Unicode standard, then the original string is returned.

See Unicode Case Mapping for implementation suggestions.

7.2.3. Formal

The formal element tells the AIML interpreter to render the contents of the element such that the first letter of

each word is in uppercase, as defined (if defined) by the locale indicated by the specified language (if specified). This is similar to methods that are sometimes called "Title Case".

<!-- Category: aiml-template-elements -->

<aiml:formal>

<!-- Content: *aiml-template-elements* -->

</aiml:formal>

If no character in this string has a different uppercase version, based on the Unicode standard, then the original string is returned.

See Unicode Case Mapping for implementation suggestions.

7.2.4. Sentence

The sentence element tells the AIML interpreter to render the contents of the element such that the first letter of each sentence is in uppercase, as defined (if defined) by the locale indicated by the specified language (if specified). Sentences are interpreted as strings whose last character is the period or full-stop character .. If the string does not contain a ., then the entire string is treated as a sentence.

<!-- Category: aiml-template-elements -->

<aiml:sentence>

<!-- Content: *aiml-template-elements* -->

</aiml:sentence>

If no character in this string has a different uppercase version, based on the Unicode standard, then the original string is returned.

See Unicode Case Mapping for implementation suggestions.

7.3. Conditional elements

AIML contains several conditional elements, which return values based on specified conditions.

7.3.1. Condition

The condition element instructs the AIML interpreter to return specified contents depending upon the results of matching a predicate against a pattern.

NB: The condition element has three different types. The three different types specified here are distinguished by an xsi:type attribute, which permits a validating XML Schema processor to validate them. Two of the types may contain li elements, of which there are three different types, whose validity is determined by the type of enclosing condition. In practice, an AIML interpreter may allow the omission of the xsi:type attribute and may instead heuristically determine which type of condition (and hence li) is in use.

7.3.1.1. Block Condition

The blockCondition type of condition has a required attribute name, which specifies an AIML predicate, and a required attribute value, which contains a simple pattern expression.

<!-- Category: aiml-template-elements -->

<aiml:condition
  xsi:type = "blockCondition"
  name = *aiml-predicate-name*
  value = *aiml-simple-pattern-expression* >

  <!-- Contents: *aiml-template-elements* -->

</aiml:condition>

If the contents of the value attribute match the value of the predicate specified by name, then the AIML interpreter should return the contents of the condition. If not, the empty string "" should be returned.

7.3.1.2. Single-predicate Condition

The singlePredicateCondition type of condition has a required attribute name, which specifies an AIML predicate. This form of condition must contain at least one li element. Zero or more of these li elements may be of the valueOnlyListItem type. Zero or one of these li elements may be of the defaultListItem type.

<!-- Category: aiml-template-elements -->

<aiml:condition
  xsi:type = "singlePredicateCondition"
  name = *aiml-predicate-name* >

  <!-- Contents: *value-only-list-item**, *default-list-item*{0,1} -->

</aiml:condition>

The singlePredicateCondition type of condition is processed as follows:

Reading each contained li in order:

1. If the li is a valueOnlyListItem type, then compare the contents of the value attribute of the li with the value of the predicate specified by the name attribute of the enclosing condition.
    a. If they match, then return the contents of the li and stop processing this condition.
    b. If they do not match, continue processing the condition.
2. If the li is a defaultListItem type, then return the contents of the li and stop processing this condition.

7.3.1.3. Multi-predicate Condition

The multiPredicateCondition type of condition has no attributes. This form of condition must contain at least one li element. Zero or more of these li elements may be of the nameValueListItem type. Zero or one of these li elements may be of the defaultListItem type.

<!-- Category: aiml-template-elements -->

<aiml:condition

xsi:type = "multiPredicateCondition">

<!-- Contents: *name-value-list-item**, *default-list-item*{0,1} -->

</aiml:condition>

The multiPredicateCondition type of condition is processed as follows:

Reading each contained li in order:

1. If the li is a nameValueListItem type, then compare the contents of the value attribute of the li with the value of the predicate specified by the name attribute of the li.
    a. If they match, then return the contents of the li and stop processing this condition.
    b. If they do not match, continue processing the condition.
2. If the li is a defaultListItem type, then return the contents of the li and stop processing this condition.

7.3.1.4. Condition List Items

As described above, two types of condition may contain li elements. There are three types of li elements. The type of li element allowed in a given condition depends upon the type of that condition, as described above.

7.3.1.4.1. Default List Items

An li element of the type defaultListItem has no attributes. It may contain any AIML template elements.

<!-- Category: condition-list-item -->

<aiml:li
  xsi:type = "defaultListItem">

  <!-- Contents: *aiml-template-elements* -->

</aiml:li>

7.3.1.4.2. Value-only List Items

An li element of the type valueOnlyListItem has a required attribute value, which must contain a simple pattern expression. The element may contain any AIML template elements.

<!-- Category: condition-list-item -->

<aiml:li
  xsi:type = "valueOnlyListItem"
  value = *aiml-simple-pattern-expression* >

  <!-- Contents: *aiml-template-elements* -->

</aiml:li>

7.3.1.4.3. Name and Value List Items

An li element of the type nameValueListItem has a required attribute name, which specifies an AIML

predicate, and a required attribute value, which contains a simple pattern expression. The element may contain any AIML template elements.

<!-- Category: condition-list-item -->

<aiml:li
  xsi:type = "nameValueListItem"
  name = *aiml-predicate-name*
  value = *aiml-simple-pattern-expression* >

  <!-- Contents: *aiml-template-elements* -->

</aiml:li>

7.3.2. Random

The random element instructs the AIML interpreter to return exactly one of its contained li elements randomly. The random element must contain one or more li elements of type defaultListItem, and cannot contain any other elements.

<!-- Category: aiml-template-elements -->

<aiml:random>

  <!-- Contents: *default-list-item*+ -->

</aiml:random>

7.4. Capture elements

AIML defines two content-capturing elements, which tell the AIML interpreter to capture their processed contents and perform some storage operation with them.

7.4.1. Set

The set element instructs the AIML interpreter to set the value of a predicate to the result of processing the contents of the set element. The set element has a required attribute name, which must be a valid AIML predicate name. If the predicate has not yet been defined, the AIML interpreter should define it in memory.

The AIML interpreter should, generically, return the result of processing the contents of the set element. The set element must not perform any text formatting or other "normalization" on the predicate contents when returning them.

The AIML interpreter implementation may optionally provide a mechanism that allows the AIML author to designate certain predicates as "return-name-when-set", which means that a set operation using such a predicate will return the name of the predicate, rather than its captured value. (See [9.2].)

A set element may contain any AIML template elements.

<!-- Category: aiml-template-elements -->

<aiml:set

name = *aiml-predicate-name* >

<!-- Contents: *aiml-template-elements* -->

</aiml:set>

7.4.2. Gossip

The gossip element instructs the AIML interpreter to capture the result of processing the contents of the gossip elements and to store these contents in a manner left up to the implementation. Most common uses of gossip have been to store captured contents in a separate file.

The gossip element does not have any attributes. It may contain any AIML template elements.

<!-- Category: aiml-template-elements -->

<aiml:gossip>

<!-- Contents: *aiml-template-elements* -->

</aiml:gossip>

7.5. Symbolic Reduction elements

AIML defines one symbolic reduction element, called srai, as well as a short-cut element called sr. The sr element is described in [7.1.7.1].

7.5.1. SRAI

The srai element instructs the AIML interpreter to pass the result of processing the contents of the srai element to the AIML matching loop, as if the input had been produced by the user (this includes stepping through the entire input normalization process). The srai element does not have any attributes. It may contain any AIML template elements.

As with all AIML elements, nested forms should be parsed from inside out, so embedded srais are perfectly acceptable.

<!-- Category: aiml-template-elements -->

<aiml:srai>

<!-- Contents: *aiml-template-elements* -->

</aiml:srai>

7.6. Transformational elements

AIML defines several transformational elements, which instruct the AIML interpreter to transform the result of processing the contents of the transformational element into another value according to a lookup table. The implementation of transformational elements is left up to the implementation.

7.6.1. Person2

The person2 element instructs the AIML interpreter to:

1. replace words with first-person aspect in the result of processing the contents of the person2 element with words with the grammatically-corresponding third-person aspect; and
2. replace words with third-person aspect in the result of processing the contents of the person2 element with words with the grammatically-corresponding first-person aspect.

The definition of "grammatically-corresponding" is left up to the implementation.

<!-- Category: aiml-template-elements -->

<aiml:person2>

  <!-- Contents: *aiml-template-elements* -->

</aiml:person2>

Historically, implementations of person2 have dealt with pronouns, likely due to the fact that most AIML has been written in English. However, the decision about whether to transform the person2 aspect of other words is left up to the implementation.

7.6.2. Person

The person element instructs the AIML interpreter to:

1. replace words with first-person aspect in the result of processing the contents of the person element with words with the grammatically-corresponding second-person aspect; and
2. replace words with second-person aspect in the result of processing the contents of the person element with words with the grammatically-corresponding first-person aspect.

The definition of "grammatically-corresponding" is left up to the implementation.

<!-- Category: aiml-template-elements -->

<aiml:person>

  <!-- Contents: *aiml-template-elements* -->

</aiml:person>

Historically, implementations of person have dealt with pronouns, likely due to the fact that most AIML has been written in English. However, the decision about whether to transform the person aspect of other words is left up to the implementation.

7.6.3. Gender

The gender element instructs the AIML interpreter to:

1. replace male-gendered words in the result of processing the contents of the gender element with the grammatically-corresponding female-gendered words; and
2. replace female-gendered words in the result of processing the contents of the gender element with the

grammatically-corresponding male-gendered words.

The definition of "grammatically-corresponding" is left up to the implementation.

<!-- Category: aiml-template-elements -->

<aiml:gender>

<!-- Contents: *aiml-template-elements* -->

</aiml:gender>

Historically, implementations of gender have exclusively dealt with pronouns, likely due to the fact that most AIML has been written in English. However, the decision about whether to transform gender of other words is left up to the implementation.

7.7. Covert elements

AIML defines two "covert" elements that instruct the AIML interpreter to perform some processing on their contents, but to not return any value.

7.7.1. Think

The think element instructs the AIML interpreter to perform all usual processing of its contents, but to not return any value, regardless of whether the contents produce output.

The think element has no attributes. It may contain any AIML template elements.

<!-- Category: aiml-template-elements -->

<aiml:think>

<!-- Contents: *aiml-template-elements* -->

</aiml:think>

7.7.2. Learn

The learn element instructs the AIML interpreter to retrieve a resource specified by a URI, and to process its AIML object contents.

<!-- Category: aiml-template-elements -->

<aiml:learn>

<!-- Contents: *uri-reference* -->

</aiml:learn>

7.8. External processor elements

AIML defines two external processor elements, which instruct the AIML interpreter to pass the contents of the elements to an external processor. External processor elements may return a value, but are not required to do so.

Contents of external processor elements may consist of character data as well as AIML template elements. If AIML template elements in the contents of an external processor element are not enclosed as CDATA, then the AIML interpreter is required to substitute the results of processing those elements before passing the contents to the external processor. As a trivial example, consider:

<system>

  echo '<get name="name"/>'

</system>

Before passing the contents of this system element to the appropriate external processor, the AIML interpreter is required to substitute the results of processing the get element.

AIML 1.0.1 does not require that any contents of an external processor element are enclosed as CDATA. An AIML interpreter should assume that any unrecognized content is character data, and simply pass it to the appropriate external processor as-is, following any processing of AIML template elements not enclosed as CDATA.

If an external processor is not available to process the contents of an external processor element, the AIML interpreter may return an error, but this is not required.

7.8.1. System

The system element instructs the AIML interpreter to pass its content (with any appropriate preprocessing, as noted above) to the system command interpreter of the local machine on which the AIML interpreter is running. The system element does not have any attributes.

<!-- Category: aiml-template-elements -->

<aiml:system>

  <!-- Contents: *character data*, *aiml-template-elements* -->

</aiml:system>

The system element may return a value.

7.8.2. JavaScript

The javascript element instructs the AIML interpreter to pass its content (with any appropriate preprocessing, as noted above) to a server-side JavaScript interpreter on the local machine on which the AIML interpreter is running. The javascript element does not have any attributes.

<!-- Category: aiml-template-elements -->

<aiml:javascript>

  <!-- Contents: *character data, aiml-template-elements* -->

</aiml:javascript>

AIML 1.0.1 does not require that an AIML interpreter include a server-side JavaScript interpreter, and does not require any particular behavior from the server-side JavaScript interpreter if it exists.

The javascript element may return a value.

8. AIML Pattern Matching

8.1. Pattern Expression syntax

An AIML Pattern Expression is a restricted form of mixed character data and optional restricted element content.

There are two forms of AIML pattern expressions: **simple pattern expressions**, and **mixed pattern expressions**.

8.1.1. Simple pattern expressions

[Definition: A **simple pattern expression** is composed from one or more simple pattern expression constituents, separated by XML spaces (#x20).]

Simple Pattern Expression

[1] SimplePattExpr ::=
SimplePattExprConst (#x20 SimplePattExprConst)+

For all simple pattern expression constituents *C*, and for all simple pattern expressions *E*, valid simple pattern expressions *V* are:	**Denoting the set of strings *L(V)* containing:**
C	all strings in *L(C)*
E	all strings in *L(E)*
C E	all strings in *L(C)* and all strings in *L(E)*
E C	all strings in *L(E)* and all strings in *L(C)*

[Definition: A **simple pattern expression constituent** is either a normal word or an AIML wildcard.

Simple Pattern Expression Constituent

[2] SimplePattExprConst ::= NormalWord | AIMLWildcard

8.1.2. Mixed pattern expressions

[Definition: A **mixed pattern expression** is composed from one or more mixed pattern expression

constituents, separated by XML spaces (#x20).]

Mixed Pattern Expression

[1] MixedPattExpr ::=
MixedPattExprConst (#x20 MixedPattExprConst)+

For all mixed pattern expression constituents *C*, and for all mixed pattern expressions *E*, valid mixed pattern expressions *V* are:	**Denoting the set of strings *L(V)* containing:**
C	all strings in *L(C)*
E	all strings in *L(E)*
C E	all strings in *L(C)* and all strings in *L(E)*
E C	all strings in *L(E)* and all strings in *L(C)*

[Definition: A **mixed pattern expression constituent** is either a normal word, an AIML wildcard, or a pattern-side bot element.]

Mixed Pattern Expression Constituent

[2] MixedPattExprConst ::=
NormalWord | AIMLWildcard | PattSideBotElem

8.1.3. Normal words

[Definition: A **normal word** consists of one or more normal characters, concatenated together.]

Normal Word

[3] NormalWord ::= NormalChar+

8.1.4. Normal characters

[Definition: A **normal character** is any "uppercase letter"*, any"caseless letter"**, or any digit.]

** for "bicameral" alphabets, in which "uppercase" has meaning (Latin, Greek, Cyrillic, Armenian, and archaic Georgian). See [Unicode Case Mapping] for more information.*

*** for "unicameral" alphabets, in which "case" has no meaning (see reference above).*

Normal Character

[4] normalChar ::=
UppercaseLetter | CaselessLetter | Digit

A normal character can be represented either as itself, or with a character reference.

8.1.5. AIML wildcards

[Definition: An **AIML wildcard** is one of * or _, which have the meanings defined in the following table.]

For all normal words *L(N)*, valid AIML wildcards *W* are:	**Denoting the set of strings *L(W)* containing:**
*	any string in *L(N)*
_	any string in *L(N)*

8.1.6. Pattern-side bot elements

[Definition: A **pattern-side bot element** consists of any AIML bot element.]

8.2. Load-time match path construction

An AIML interpreter must construct a **match path** from each category at load-time. It is recommended, but not required, that all match paths be compressed into an efficient trie structure such as the Graphmaster (see [8.4.1]).

A match path has three components, whose order is mandatory:

1. **pattern**: the contents of the category element's pattern element
2. **that**: the contents of the category element's pattern-side that element

   NB: If the category does not contain a pattern-side that, then the AIML interpreter must insert an "implied" that match path component with value * (single asterisk wildcard).
3. **topic**: the contents of the name attribute of the topic element parent of the category element.

   NB: If the category element does not have an explicit topic parent, then the AIML interpreter must insert an "implied" topic match path component with value with value * (single asterisk wildcard).

A conventional way to construct the match path is as a simple string, in which the markers <that> and <topic> are used to separate the pattern, that and topic components from one another. For example, given this topic/category:

```
<topic name="* A">
  <category>
    <pattern>_ C *</pattern>
    <that>B *</that>
    <template>
      ...
    </template>
```

```
  </category>
</topic>
```

the match path string might look like:

_ C * <that> B * <topic> * A

(The convention of using angle-bracketed markers such as <that> and <topic> is not to be confused with XML markup.)

Following the same convention, and by way of illustrating the requirement of inserting "implied" that and/or topic components where they are not explicit, here are three more examples.

Given this topic/category:

```
<topic name="* A">
  <category>
    <pattern>_ C *</pattern>
    <template>
      ...
    </template>
  </category>
</topic>
```

the match path string would look like ("implied" that):

_ C * <that> * <topic> * A

Given this category:

```
<category>
  <pattern>_ C *</pattern>
  <that>B *</that>
  <template>
    ...
  </template>
</category>
```

the match path string would look like ("implied" topic):

_ C * <that> B * <topic> *

And given this category:

```
<category>
  <pattern>_ C *</pattern>
  <template>
    ...
  </template>
</category>
```

the match path string would look like ("implied" that *and* topic):

_ C * <that> * <topic> *

At run-time, an AIML interpreter must match normalized inputs against match paths that are ordered as described in [8.4].

8.3. Input normalization

An AIML interpreter must perform a "normalization" function on all inputs before attempting to match. The minimum set of normalizations is called **pattern-fitting normalizations**. Additional normalizations performed at user option are called **sentence-splitting normalizations** and **substitution normalizations** (or just "substitutions").

If an AIML interpreter performs substitution normalizations on the input, then these must be performed first.

If an AIML interpreter performs sentence-splitting normalizations on the input, then these must be performed on the output of the substitution normalization process.

The pattern-fitting normalization process receives the output of the sentence-splitting normalization process (if any), or the output of the substitution normalization process (if any, and if no sentence-splitting normalization is performed), or the direct input (if no sentence-splitting or substitution normalization is performed).

8.3.1. Substitution normalizations

Substitution normalizations are heuristics applied to an input that attempt to retain information in the input that would otherwise be lost during the sentence-splitting or pattern-fitting normalizations. For example:

- Abbreviations such as "Mr." may be "spelled out" as "Mister" to avoid sentence-splitting at the period in the abbreviated form
- Web addresses such as "http://alicebot.org" may be "sounded out" as "http alicebot dot org" to assist the AIML author in writing patterns that match Web addresses
- Filename extensions may be separated from their file names to avoid sentence-splitting (".zip" to " zip")

8.3.2. Sentence-splitting normalizations

Sentence-splitting normalizations are heuristics applied to an input that attempt to break it into "sentences". The notion of "sentence", however, is ill-defined for many languages, so the heuristics for division into sentences are left up to the developer.

Commonly, sentence-splitting heuristics use simple rules like "break sentences at periods", which in turn rely upon substitutions performed in the substitution normalization phase, such as those which substitute full words for their abbreviations.

8.3.3. Pattern-fitting normalizations

[Definition: **Pattern-fitting normalizations** are normalizations that remove from the input characters that are not normal characters.]

Pattern-fitting normalizations on an input must remove all characters that are not normal characters. For each non-normal character in the input,

- if it is a lowercase letter, replace it with its uppercase equivalent
- if it is not a lowercase letter, replace it with a space (#x20)

8.3.4. Normalization examples

Since input normalization can have any of 1, 2 or 3 stages in a given AIML interpreter, results of normalization may vary widely among different AIML interpreters. Here we show examples of input normalization in a typical English-language-oriented AIML interpreter that includes the full complement of normalizations.

input	substitution normalized form	sentence-splitting normalized form	pattern-fitting normalized form
"What time is it?"	"What time is it?"	"What time is it"	"WHAT TIME IS IT"
"Quickly, go to http://alicebot.org!"	"Quickly, go to http://alicebot dot org!"	"Quickly, go to http://alicebot dot org"	"QUICKLY GO TO HTTP ALICEBOT DOT ORG"
":-) That's funny."	"That is funny."	"That is funny"	"THAT IS FUNNY"
"I don't know. Do you, or will you, have a robots.txt file?"	"I do not know. Do you, or will you, have a robots dot txt file?"	"I do not know" "Do you, or will you, have a robots dot txt file"	"I DO NOT KNOW" "DO YOU OR WILL YOU HAVE A ROBOTS DOT TXT FILE"

8.4. Pattern Expression matching behavior

Each input to the AIML interpreter must pass through the input normalization process described above, in which (at the very minimum) the input will be processed according to the description of pattern-fitting normalizations.

In the case that sentence-splitting normalization is used by the AIML interpreter, a single input may be subdivided into several "sentences". The AIML interpreter must process each sentence of the input by producing an **input path** from it.

An input match path has three components, whose order is mandatory, and which correspond to the three

components of a load-time match path:

1. **pattern**: the normalized input
2. **that**: the previous bot output, normalized according to the same rules as in input normalization.

   NB: If there was *no* previous bot output, or the previous bot output was unavailable, the value of the input path that is *.
3. **topic**: the current value of the topic predicate.

   NB: If the topic predicate has *no* value, then the value of the input path topic is *.

A conventional way to construct the match path is as a simple string, in which the markers <that> and <topic> are used to separate the pattern, that and topic components from one another. (The convention of using angle-bracketed markers such as <that> and <topic> is not to be confused with XML markup.)

Here are some examples of input paths that would be constructed, according to this convention. (Paths are shown inside quotes for clarity, but actual paths do not include quotation marks.)

normalized input	previous bot output (normalized)	value of topic predicate	input path
"YES"	"DO YOU LIKE CHEESE"	""	"YES <that> DO YOU LIKE CHEESE <topic> *"
"MY NAME IS NOEL"	"I GUESS SO"	"MUSHROOMS"	"MY NAME IS NOEL <that> I GUESS SO <topic> MUSHROOMS"

Each "sentence" of the input is matched word by word, pattern by pattern, against the total set of loaded match patterns. The total set of loaded match patterns is at least partially ordered, so that:

1. Each _ wildcard comes before words that begin with any other character (including pattern-side bot elements).
2. Each * wildcard comes *after* words that begin with any other character (including pattern-side bot elements).

As soon as the first complete match is found, the process stops, and the template that belongs to the category whose pattern was matched is processed by the AIML interpreter to construct the output.

8.4.1. Explanation via implementation description: Graphmaster

Matching behavior can be described in terms of the class Graphmaster, which is a common implementation of the AIML pattern expression matching behavior:

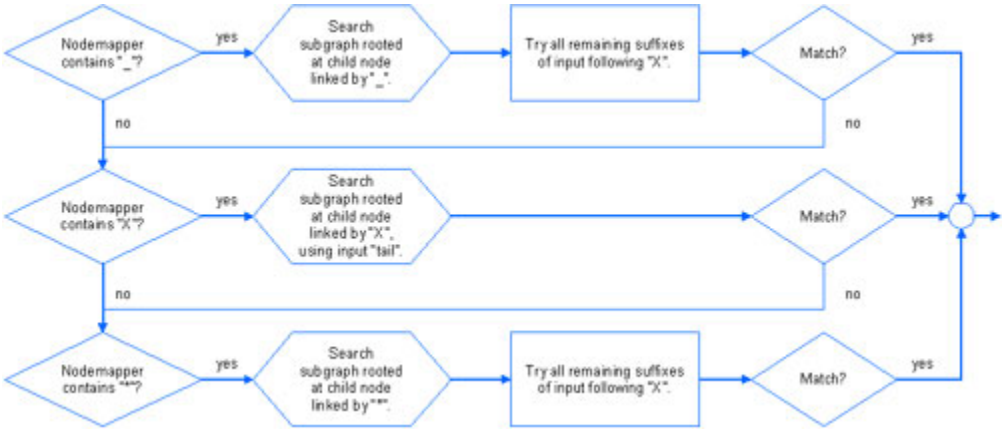
1. Given:
   a. an input starting with word X, and
   b. a Nodemapper of the graph:
2. Does the Nodemapper contain the key _? If so, search the subgraph rooted at the child node linked by _. Try all remaining suffixes of the input following X to see if one matches. If no match was found, try:
3. Does the Nodemapper contain the key X? If so, search the subgraph rooted at the child node linked by X, using the tail of the input (the suffix of the input with X removed). If no match was found, try:

4. Does the Nodemapper contain the key *? If so, search the subgraph rooted at the child node linked by *. Try all remaining suffixes of the input following X to see if one matches. If no match was found, go back up the graph to the parent of this node, and put X back on the head of the input.
5. If the input is null (no more words) and the Nodemapper contains the <template> key, then a match was found. Halt the search and return the matching node.

If the root Nodemapper contains a key "*" and it points to a leaf node, then the algorithm is guaranteed to find a match.

Note that:

1. The patterns need not be ordered alphabetically or according to any other complete system, only partially ordered so that _ comes before any word and * after any word.
2. The matching is word-by-word, not category-by-category.
3. The algorithm combines the input pattern, the <that> pattern, and the <topic> pattern into a single "path" or sentence such as: "PATTERN <that> THAT <topic> TOPIC" and treats the tokens <that> and <topic> like ordinary words. The PATTERN, THAT and TOPIC patterns may contain multiple wildcards.
4. The matching algorithm is a highly restricted version of depth-first search, also known as backtracking.



9. AIML Predicates

[Definition: An AIML **predicate** is an item that can be "declared" at any point in an AIML object and whose value can be manipulated by the AIML object within template elements.]

9.1. AIML Predicate name syntax

[Definition: An AIML **predicate name** is a string composed of one or more normal characters, concatenated together.]

Predicate Name

[1] predName ::= normalChar+

9.2. AIML Predicate behaviors

Generically, when a set element is used to set the value of a predicate, the return value of the set is the result

of processing the contents of the set.

However, an AIML interpreter may provide a mechanism for designating predicates as "return-name-when-set". This means that a set operation using such a predicate will return the name of the predicate, rather than its captured value.

This feature is meant to allow concise inline predicate-setting and "natural" conversation. The following two categories show a simple example of this usage:

```
<category>
  <pattern>HE DID IT</pattern>
  <template>
    Who?
  </template>
</category>
<category>
  <pattern>*</pattern>
  <that>WHO</that>
  <template>
    Oh, why do you think <set name="he"><star/></set> did that? I wouldn't expect that kind of behavior from <get name="he"/>.
  </template>
</category>
```

If the predicate "he" has been designated as "return-name-when-set", then the following dialogue could take place:

user> He did it.

bot> Who?

bot> Joe.

bot> Oh, why do you think he did that? I wouldn't expect that kind of behavior from Joe.

Currently, AIML does not provide a mechanism for specifying predicates as "return-name-when-set" from within AIML.

9.3. AIML Predicate defaults

AIML interpreters may optionally provide a method for setting "default" values for predicates, so that if the

predicates are called using a get element but have not been specially defined within a category that has yet been activated, they will return a standard value.

AIML does not currently provide a mechanism for specifying default values of predicates from within AIML.

10. AIML Schema

A complete version of this document will include an XML Schema for AIML.

11. References

(This list is incomplete.)

- AIML 1.0 Tags Set (http://alicebot.org/committees/architecture/resolutions/aiml10.html)
- AIML Specification Release Process (http://alicebot.org/committees/architecture/resolutions/aiml-specification-process.html)
- Extensible Markup Language (XML) 1.0 (Second Edition) (http://www.w3.org/TR/REC-xml)
- Namespaces in XML (http://www.w3.org/TR/REC-xml-names/)
- Unicode Case Mapping (http://www.unicode.org/unicode/reports/tr21/)
- XSL Transformations (XSLT) Version 1.0 (http://www.w3.org/TR/xslt)

12. Version History

- WD-001: Initial version of this document (Noel Bush).
- WD-002: Small corrections made (Noel Bush)
  - replaced ambiguous "AIML Pattern Expression" with "mixed pattern expression" or "simple pattern expression" as appropriate in several places.
  - removed confusing "into several different categories" phrase from intro text to 7
  - changed "must return a particular value" to "must return a value according to the functional meaning of the element" in 7.1
  - appended "for the second dimension" in 7.1.2. Template-side That, to clarify meaning of unspecified second dimension of index attribute
  - changed "an empty string" to "the empty string "" in 7.1.6
  - appended "with an AIML predicate name" for clarification in 7.1.6
  - inserted heading text for cross-reference to sr in 7.5
  - changed grammar in 8 to say "simple pattern expressions" instead of "a simple pattern expression", "mixed pattern expressions" instead of "a mixed pattern expression"
  - inserted missing "mixed" qualifier for several terms in the table in 8.1.2
  - added "In practice, bot applications often strip out 'punctuation' characters, but this is not required." to 8.1.4
  - changed two instances of "*L(S)*" to "*L(N)*" to make sense in the table in 8.1.5
- WD-003: changed URI to reference to Word-generated HTML version, temporarily using this facility for quick availability on Web site
- WD-004: Reformatting and corrections (Noel Bush)
  - reformatted Word-generated version to good HTML
  - corrected 7.6.3 to actually use gender element (thanks Tom Ringate)

    - included some reference hyperlinks within the text
    - changed "an AIML object may contain zero or more..." to "an AIML object must contain zero or more..." ("may" to "must" in 3.2 (AIML Element)
    - added some links to sentence-splitting normalization description at various places in document
- WD-005: Corrections (Noel Bush)
    - changed "AIML 1.0.1" to "AIML 1.0.1" to match ArchComm's spec release process requirements
    - changed language in Status section to emphasize that this document is not final
    - removed if per ArchComm resolution.
    - corrected Pattern-side that example (thanks Sherman Monroe)
    - added language to 7.1.6 (Get) describing optional implementation of default-value mechanism for predicates
    - added language to 7.4.1 (Set) describing optional implementation of "return-name-when-set" support for predicates
    - added language to 7.5.1 (SRAI) emphasizing the validity of embedded srais (thanks Sherman Monroe)
    - changed use of word "entities" to "units" or "elements" in a few cases to avoid misperception that we're talking about other kinds of entities (thanks Sherman Monroe)
    - removed incomplete idea of "Fallback" from 3.3 (Forward-Compatible Processing)
- WD-006: Corrections and Amendations (Noel Bush)
    - fixed sentence in 3.2 to say "an AIML interpreter must" instead of "an AIML must" (thanks Andrew Teal)
    - touchup on some definitions in section 2
    - significant amendation of 2.10 (White Space Handling) to reflect common assumptions about behavior of AIML interpreters with respect to white space, and to include ability for AIML objects to control this behavior
    - revised namespace URI to include version number :-(
    - changed example of "abbreviations...in support of historical usage" in 3.1 from <srai> to <li> as a better specimen of what we mean by "historical"
    - various additions of hyperlinks to key references
    - changed definition of "predicate" in 3.5 to omit the word "variable" and use generic "item" instead (historical baggage)
    - changed "bot properties" to "bot predicates" in various places; added explicit statement in 3.5 that their values are set at load time
    - corrected definition of Pattern-side That (6.1) to omit language about attributes (has not been approved by ArchComm); also corrected explanation of how that contents are used, and added description of default value
    - renamed 8 (AIML Pattern Expressions) to "AIML Pattern Matching"
    - removed incorrect statement in 8.1.4 that removing punctuation is not required, and amended definition of "normal characters"
    - formatting and link cleanup in pattern expression definitions
    - renumbered 8.2 (Pattern Expression matching behavior) to 8.4 and 8.2.1 (Graphmaster implementation example) to 8.4.1, and inserted new 8.2 (Load-time match path construction) and 8.3 (Input normalization) with subsections 8.3.1 (Substitution normalizations), 8.3.2 (Sentence normalizations), 8.3.3 (Pattern-fitting normalizations) and 8.3.4 (Normalization examples) (significant new description)
    - changed name of 3 from "AIML Predicate Names" to "AIML Predicates"; moved 9 (AIML Predicate name syntax) to 9.1, and made 9 "AIML Predicate handling"; added 9.2 (AIML Predicate behaviors) and 9.3 (AIML Predicate defaults)

    - softened language in 7.1.6.1 (Bot) to say that bot *may be considered* (emphasis not in text) a restricted version of get, rather than that it "is"
- WD-007: Correction (Richard Wallace)
    - Reversed descriptions of <person> and <person2>. <person> came from the original ELIZA program was originally intended to swap first and second person (*I* and *You*). The operator <person2> was numbered 2 becase it was added second, even though it swaps 1st and 3rd pronouns (*I* and *he* or *she*).
- WD-009: Final Draft adopted by ALICE A.I. Foundation
    - This document adopted as the official AIML 1.0.1 standard document by the ALICE A.I. Foundation