Bot Libre Artificial Intelligence Engine

Overview

This is a high level design document for the Bot Libre AI engine. The Bot Libre AI engine is a Java class library designed for natural language processing, communication, event processing, and machine learning. It can be used from a Java application, such as a Java web server, Android Java application, or Java desktop application.

Bot Libre uses an object oriented design that is modeled after the human brain.

The Bot Libre AI engine provides the following services:

- Natural language processing
- Database memory
- File memory
- Emotions
- Vision processing
- Machine learning
- Event processing
- State machines
- AIML scripting
- Self scripting
- Response lists
- Chat
- Facebook
- Twitter
- Telegram
- Slack
- Skype
- IRC
- Email

- Twilio SMS
- Google API
- WikiData
- Wiktionary

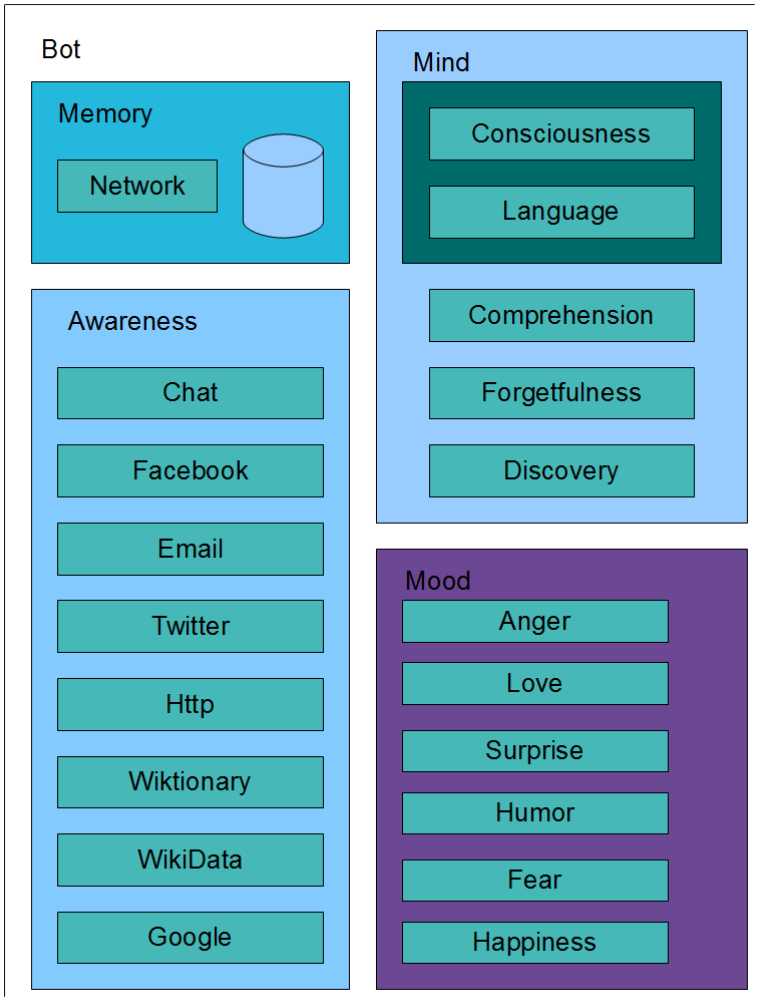
Components

Bot Libre defines a set of high level interfaces that model a “Bot”. A bot is a component that defines an instance that is composed of a memory, mind, mood, and awareness.

The bot's memory can either be stored in a database, or in a file.

The bot's awareness is a collection of sense objects. A sense object defines how the bot can interact, such as Chat, Facebook, Email, or others.

The bot's mind is defined by several thought objects, such as language processing, and others.



Memory

Bot Libre provides three **Memory** implementations, **DatabaseMemory**, **SerializedMemory,** and **MicroMemory**.

The bot's memory is normally stored in a relational database. PostgreSQL is normally used for the bot's database, but other JDBC compliant databases such as Derby are supported. The EclipseLink JPA library is used by the DatabaseMemory class to access the bot's database. Each bot has its own isolated database (or database schema).

The SerializedMemory can also be used to store the bot's memory to a Java serialization file. The SerializedMemory is meant for small or embedded bots, and does not scale beyond a certain size.

The MicroMemory is similar to the SerializedMemory but uses an optimized binary file format. It is much faster than the SerializedMemory and optimized for Android devices.

All of the bot's data is stored in its memory. This includes its responses, scripts, settings, and other data. The bot's databas is composed of two main classes and tables, **Vertex** and **Relationship**. Vertex and Relationship define an object oriented meta-model to define all of the bot's data.

Each Vertex has a set of relationships, each relationship has a *source*, *type*, *meta*, and target, all of which are other Vertex objects.

Classes:

- Memory : DatabaseMemory, SerializedMemory, MicroMemory
- Network : BasicNetwork, DatabaseNetwork
- Vertex : BasicVertex
- Relationship : BasicRelationship
- Primitive
- Property
- BinaryData
- TextData

Mind

The bot's **Mind** is a multi threaded processing model for processing events such as natural language input. The mind has a set of **Thought** objects that process input and send output to the senses.

There are two types of thought objects, conscious thoughts, and subconscious thoughts. Conscious thoughts are processed on the mind's main thread sequentially. Subconscious thoughts are processed on a background thread.

The main conscious thought classes are **Language** for natural language processing, and the **Consciousness**.

The subconscious thoughts include **Discovery**, **Comprehension**, and **Forgetfulness**.

The Language class performs the main processing for the bot. It takes the input from the memory's active input queue, and processes the input using the bot's scripts, and a heuristic response matching algorithm.

Classes:

- Mind : BasicMind
- Thought : Language, Consciousness
- SubconsciousThought : Discover, Comprehension, Forgetfulness

Awareness

The bot's **Awareness** defines a common interface to sending and receiving input. Input can be a chat message, or a command or event. The awareness has a set of **Sense** objects that support various communications mechanisms such as chat, email, social media, web services, and vision.

Most sense classes define an interface into an external API, such as **Facebook**, **Twitter**, **Telegram**, **WikiData**, and others. The sense object converts the bot's input objects to and from the data format of the external API, such as converting the objects to XML or JSON, and processing HTTP requests.

Classes:

- Awareness : BasicAwareness
- Sense
  - Chat
  - Facebook
  - Twitter
  - Telegram

- Slack
- Skype
- Email
- IRC
- Twilio
- Http
  - Google
  - WikiData
  - Wiktionary
- Context
- RemoteService
- TextEntry

Scripting

Bot Libre provides support for multiple scripting languages and response formats including **Self**, **AIML**, **response lists**, and **chat logs**.

All of the bot's responses and scripts are stored in its memory. Scripts are parsed and compiled into state and function objects. Response lists and chat logs parsed into question and response objects. Everything is stored as vertex and relationship objects in the bot's object oriented memory.

Self

Self scripts are compiled using the **Self4Compiler** or **Self4BytecodeCompiler** classes. The Self code is parsed and compiled into either state and function objects, or byte code (which is later parsed into state and function objects). The Self4BytecodeCompiler is normally used as it stores an entire state or function into a single binary data object, instead of one object per operation (so takes much less database access to load).

Once a script is loaded as state and function objects it is processed as a state machine by the Language class, with the help of the SelfInterpreter class.

AIML

AIML is parsed from XML and translated to Self code. AIML is stored and executed as Self state, pattern, and template objects.

AIML can also be loaded as response objects. When loaded as response objects the response heuristic is used to find matching patterns.

The AIML templates are always translated to Self templates. Self includes operations that mirror most AIML template tags.

Response Lists

Response lists and chat log files are parsed and converted to question and response objects. Question and response objects are stored in the bot's memory as vertex and relationship objects, as is all data.

The question object's *#response* relationship uses the relationship *meta* object to store meta-data about the response, such as the topic, keywords, and required words.

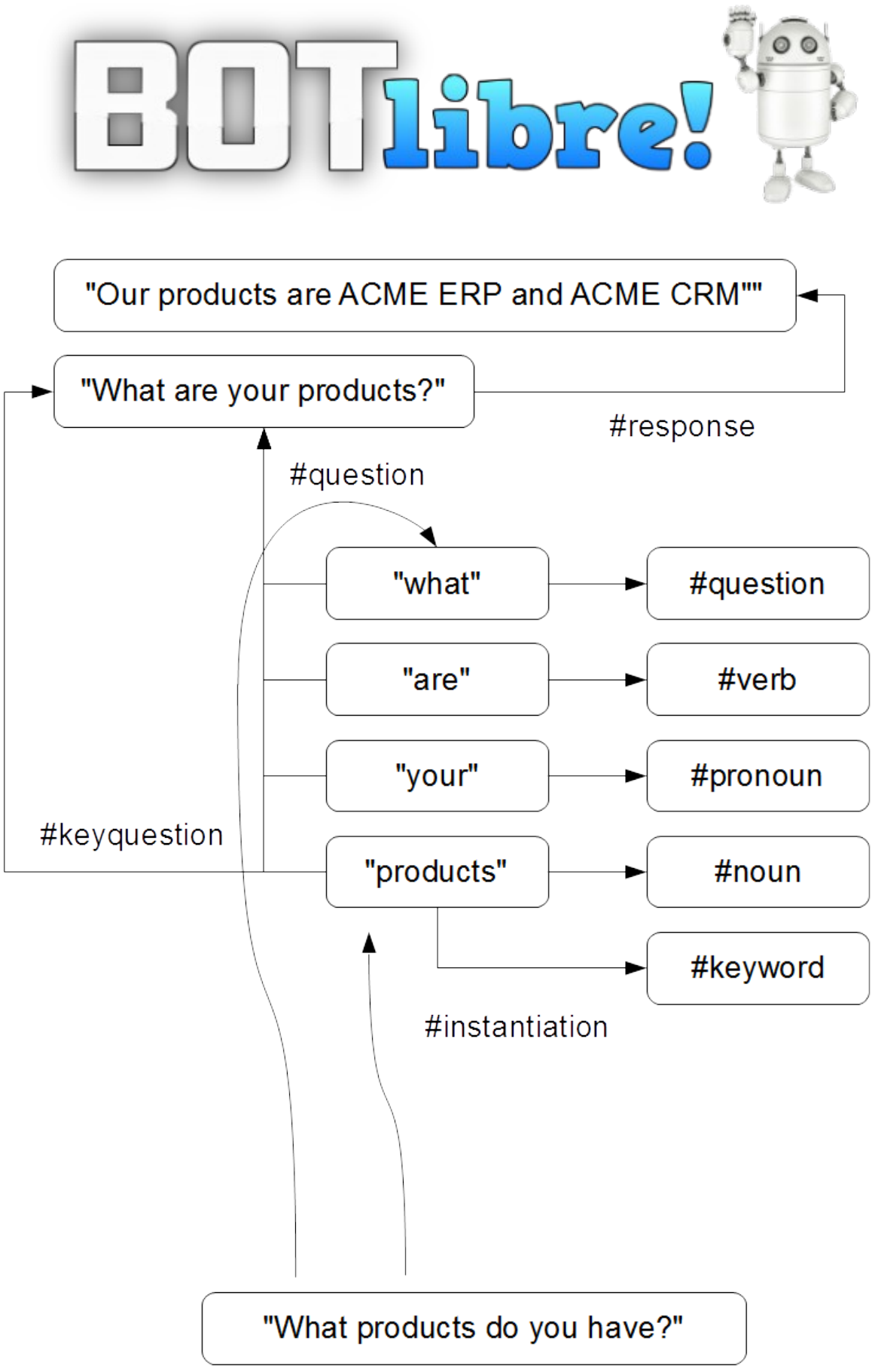
All question objects are split into their words, and each word is stored as its own object and related to the question. This well connected design allows the Language class to heuristically find similar matching questions using word relationships, similar to how the human brain works.

Natural Language Processing

Bot Libre provides a heterogeneous NLP architecture that supports multiple NLP mechanisms. Bot Libre supports NLP through keyword heuristics, patterns, state machines, and scripting.

Keyword Heuristic

Bot Libre's primary NLP mechanism is its keyword heuristic. This allows the bot to find the closest matching trained question to the user's input and return the best response for the context with minimal training and no programming.


BOTlibre!
"Our products are ACME ERP and ACME CRM""
"What are your products?"
#response
#question
"what"
#question
"are"
#verb
"your"
#pronoun
#keyquestion
"products"
#noun
#keyword
#instantiation
"What products do you have?"

The heuristic algorithm is directly related to the way Bot Libre stores knowledge in its object database.

When a phrase is input into a bot from a user input, or from a trained response list question and response, it is parsed into a list of words. In addition to the phrase, each word is stored in the bot's knowledge base. The phrase is related to its words, and for a trained question, each word is related back to its question.

All question words are related back to their question using the #question relationship. So each word will have a relationship to all questions that it is contained in.

If a response had a keyword defined in its meta-data, the keyword would also be related to its question using the #keyquestion relationship.

When a bot is given an input phrase it first checks if it has any trained responses for that exact question. If it does, it returns the best matching response for the context.

If there is no exact match, then the bot will search for the closest matching trained question. To find the question it will parse the input phrase into its words, and for each word it will lookup all trained questions for that word using its #question relationship.

Each word is given a score depending on the type of word, keywords are scored highest, nouns and adjectives are scored higher than verbs, and articles are scored the lowest. The total score for each possible matching question is calculated and the best match is used.

If the best match score is not sufficient (normally 50% of the max score), then the bot will use its default response.

The best match may not be valid for the context, such as a topic, previous, required word, or other condition to matching. If the best match is not valid, then the next best match will be checked until a valid match if found. If a response has a keyword that matches the user's input then the response is considered valid, no matter what the % match is.

The default response is used when there is no valid match found. If the bot has no default response, then it just mimics the user's input back, or uses language synthesis.

The default response also considers the context, and if there is a topic, previous, or other condition to a default response it will be used over a default response with no condition.

Pattern Keyword Heuristic

The keyword heuristic can also be used with patterns. When the bot is trained on a pattern question and response, the pattern's words are indexed the same as a phrase question's words. Pattern words use the #pattern relationship instead of the #question relationship.

The lookup algorithm for a pattern is the same as for a phrase question. The difference is that the #pattern relationship is used to find patterns, instead of the #question relationship. Also the pattern is evaluated to determine if it matches, instead of requiring a 50% match.

This allows for a hybrid NLP approach using keywords and patterns, and results in finding the closest matching pattern consider all of the words, instead of finding the pattern that has the first matching words that a pattern tree will result in.

Patterns loaded as responses result in this behavior. Patterns and AIML can also be loaded as scripts, which results in different behavior. When loaded as a script, a pattern uses a pattern tree, so the pattern with the first word that matches will be used, instead of the pattern that matches the most words (for standard AIML compliance, AIML should be loaded as scripts).

Patterns

A pattern is an expression that can be evaluated for a phrase and returns either true or false depending on if it matches or not. Patterns are very different than the bot's heuristic as they either match or do not, they are not a fuzzy algorithm like the heuristic.

This has the draw back of requiring the bot developer to consider every possible pattern of words the user may use, but have the benefit of being more precise. Normally it is recommend to use patterns for scripted responses, and to use response lists for general questions.

AIML patterns are parsed into a pattern tree using state machines. To process a user input each word in the input is processed through the bot's state machines to arrive at a pattern. The pattern is then evaluated, and if true, it's template is executed.

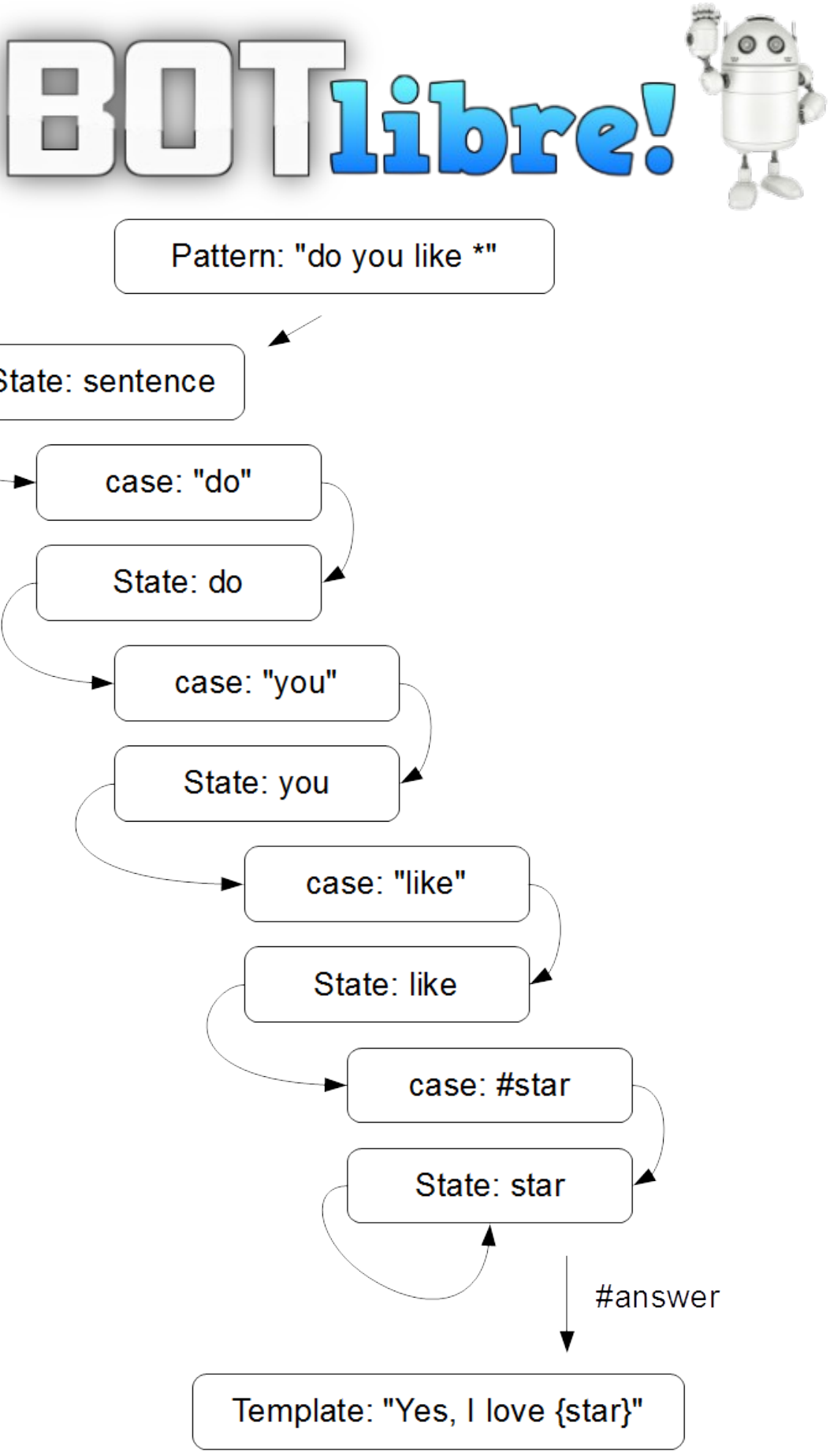
Bot Libre patterns support the AIML pattern syntax, but also merge some pattern features from ChatScript, and also can include Self code.

Patterns support:

- *, _, ^, # wildcards
- [word sets] and (optional) words
- variables and {self code}

State Machines

A state machine processes an input and transitions to the next state. State machines are good at processing logical language and input. The bot's Language class stores a set of state machines compiled from Self or AIML scripts.


BOTlibre!
Pattern: "do you like *"
State: sentence
case: "do"
State: do
case: "you"
State: you
case: "like"
State: like
case: #star
State: star
#answer
Template: "Yes, I love {star}"

The user's input is passed to each state machine. The state machine processes the input and either returns a response result, or null if no state match is found.

Each state machine starts with a main state. The main state can define a set of patterns, or a set of cases that goto child states. Typically the main state will loop over the input's words and process each word in its child states.

State machines allow for complex language to be parsed similar to a compiler parsing a programming language. State machines can be used to process complex mathematical expressions, or natural language.

Both AIML and Self are compiled to state machines. AIML patterns are compiled to state machines to process their semantic precedence ordering, but AIML does not have a concept of state machines, so the advanced features of state machines cannot be used from AIML.

Self includes a concept of state objects as a first class artifacts similar to functions. Self allows for sophisticated state machines to be defined.

Scripting

Natural language can also be processed using scripting written in Self. Using a case or star pattern the entire user input can be passed to a Self function. The function can use regular programming to parse and process an input and return a response.

Self also allows for web services to be called. This allows for third party web services to be used to process and input and determine a response.