AI Programming Language

blame: Matt Taylor

A simple programming language for AI Agent programming.

Agents use LLMs with sophisticated system prompts that are rendered in 'real time'.

The AI Programming Language (AIPL) is a means by which to efficiently manipulate system prompts as an Agent conversation evolves over time.

A simple mustache-like case-insensitive templating language is used for rendering variables for prompt-templates. These rendered prompt templates are used within the live chat context via system-messages.

AIPL provides for a set of known events, and a simple syntax for updating state variables as the conversation is happening. This real-time updated state can be used inside system message prompts to update the context, similar to how one might query a vector database for RAG.

There is a balance in AIPL between 'hard code' (where syntax rules must be obeyed and the outcome is predictable) and 'soft code' (where the LLM is used to interpret programmer intent or answer questions, and the outcome is creative and somewhat unpredictable)

AIPL allows the Agent programmer the ability to create an Agent that can respond to the user in different *controllable* ways, depending on how a conversation evolves with the user, or other outside events that happen during the course of the conversation.

State, Events, Logic

State, events, logic handling and syntax are the core aspects of any programming language. AIPL is no different. Unlike traditional programming languages however it strives to utilize the power of LLMs as much as possible to 'flesh out' intentions of the developer where it makes sense to do so.

State

State is captured in as series of `ChatData` values associated with a given conversation of the form:

```
type ChatData = {
  namespace: string; // namespace of the data
  type: string; // type of data
  value: Record<string, string> // key/value pairs of data
}
```

- State is global and mutable within the conversation
  - All ChatData objects are stored within the conversation and ordered by creation time. The last created ChatData for a given namespace is used within prompt templates.
- State is referened inside a prompt-template as `{namespace.value}`
  - all values are accessed via namespace and key

    - all values are natively stored as strings
- `ChatData.type` is used as a system-level hint to determine where and how to use or display the data outside the context of the conversation.

Events Updating State

Events are emitted at various defined points of an ongoing conversation. Events follow the URN convention of `:` delimiters. Events have a specific firing order, and are fired in the natural sequence order of the conversation. Outside events may also be called depending on the execution environment of the Agent that will influence the next utterance of the Agent, or possibly induce an agent to make an utterance without user interaction.

"Just a moment. I've just picked up a a fault in the AE-35 unit..."

Events give opportunities to use and update ChatData and are the main hook by which AIPL code is evaluated.

The basic AIPL event handler syntax follows the forms:

```
event(condition)? state-location -> state-question
```

```
event? state-location -> state-question
```

State questions are simple questions asked to the LLM given the context of the conversation message history, excluding any other system messages. The answers to these questions are stored in the ChatData states. The event will only fire and update the ChatData state if the condition is evaluated by the LLM to be true or if no condition is present.

Update ChatData State Example

```
chat:message:assistant(happy({mood.recent})) ? cookie.shareCount -> How
many cookies should {assistant} share with {user}?
```

Note that AIPL assumes an LLM with sufficient power to answer the question:

```
How would the following function evaluate if it MUST return true or false?

happy("John is upset at Diane after she stole his milk")
```

NOTE that these LLM driven 'soft code' evaluations are provided WITHOUT full conversation context, with the expectation that they could be evaluated on multiple LLM inferences *simultainiously* with a majority-wins style strategy to improve reliability.

AIPL leverages the power of the LLM to be creative in its responses and interperate the *meaning* of function names without prior definition -or- with some minimal natural language definition if provided.

This mixture of 'hard' and 'soft' code allows for fast development as no 'real code' is ever written for 'soft code'. This is especially useful for what might be a difficult nuanced evaluation of the present state of a conversation. As LLMs improve over time it is expected that the same code would become more and more reliable.

Prompt Template

A simple mustache-inspired forgiving syntax style. Note that either `{namespace.key}` or `{{nameSPACE.KeY}}` are valid and equivalent.

If no default value is present then undefined values are rendered as an empty string.

Default values are optional and are declared via a colon character after the variable like `{namespace.key:default value}`.

Only alpha-numeric characters are allowed for namespaces or keys following the regex pattern: `[a-zA-Z0-9]`.

The following template variables are special system provided variables:

{char} = the character/agent name attached to the prompt template {user} = the current user role chat participant name {assistant} = the current assistant role chat participant name {date} = the current date {time} = the current time

Prompt Template Example:

```
{char}'s mood is {mood.recent:happy enough} and they currently have
{cookie.count:12} cookies. They are aware of the following dangers:
{environment.dangers}

{char} is aware that the current date is {date} and the current time is
{time}
```