AI Programming Language

blame: Matt Taylor

previous versions

current version: 0.3

A simple programming language for LLM driven AI Agents.

Definition of terms

LLMs (Large Language Models) are *effectively* AI (Artificial Intelligence) as most people imagine and use that term.

Agents are LLMs personified.

The chat/message-passing paradigm for intereacting with LLMs is currently the best mechanism for interacting with LLMs. This 'chat' style and allows users to have intelligent thoughtful conversations with Agents, or give direction to Agents to perform tasks, allowing the Agent to have a sense of the conversational context.

The industry has standardized on three main 'roles' for each message: `system`, `user`, and `assistant`.

The `user` and `assistant` role messages are self describing. The `system` role messsage is the message responsible for giving direct instruction to the LLM.

There are typically one or more system messages within the context of the conversation.

System message content can be as simple as 'You are a helpful AI that answers all questions truthfully' or as complex as semi-structures RAG query results the LLM should use for answering questions. The only true limits are those imposed by context size (how many tokens the LLM can use) and the power of the LLM to correctly understand and interpret the information.

Why

Agents are uncanny in their ability to adopt a given personality if it is described well enough. LLMs in a way are the ultimate mimics.

This mimicry is often good enough for simple conversations and tasks as the user will provide the LLM with enough motivation and conext in their messages they send through the 'chat window' for the LLM to respond in a reasonable manor.

The problem comes when the Agent author (the one who crafted the Agent) wants to be in more control of the Agent.

Often the Agent author will want the Agent to behave in different ways depending on how the conversation progresses.

Perhaps the Agent is to be a salesperson, and the author has a clear idea of how a salesperson working on their behalf should go about the business of selling.

In the beginning stages of the conversation the author wants to insure that the Agent follows a kind of 'sales script' as many human salespeople currently follow.

Perhaps the sales script looks like:

- Greet the prospect
- Gather information about how the prospects day is going, and if they have a favorite sports team
- Gather information on what the prospect's needs are
- Communicate how the Agent's employer can help with their needs.
- Setup a future meeting with a specific time and date of the prospects choosing within a certain availability window.

Today with standard prompt engineering it is possible to create an Agent that performs some of these tasks but likely not all.

Agents will often 'wonder' and stray from the Agent authors intent during the conversation as the only feedback the Agent has during the conversation is from the user, NOT the Agent author.

AIPL attempts to address this imbalance, and give the Agent author more of a direct presense and better control as the conversation is happening. This is achieved by giving the Agent author the tools to update the Agent as the conversation progresses.

What

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 a simple syntax for updating state variables and executing conditional logic as the conversation is happening. This real-time updated state can be used inside system message prompts to update the chat context, to provide additional information and guidance, similar to how one might query a vector database for RAG, or change the Agents personality or goals by updating the system messages that describe these details.

There is a balance in AIPL between `traditional 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 author 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 values associated with a given conversation. It can be read from as template variables, and can be updated by calling out to the LLM to answer questions, or evaluate 'soft code'.

- State is global and mutable within the conversation
  - All state is stored within the conversation and ordered by creation time. The last stored value within a given namespace/key pair is used within prompt templates.
- State is referened inside a prompt-template as `{namespace.key}`
  - all values are accessed via namespace and key
  - all values are natively stored as strings
  - values can be interpreted within the language as a string, number, or boolean depending on context, by asking the LLM how the data should be coerced.

Updating State

The basic AIPL state update syntax follows the forms:

update the state:

`(expression -> identifier)`

`identifier` is the full name of the state variable including namespace and key.

`expression` follows c-style expression semantics where boolean algebra and comparison operators are available.

Quoted strings act as 'natural language soft code' for the LLM to evaluate and attempt to answer, and are evaluated given the full context of the chat history during direct assignment.

'soft code' evaluation in the context of a boolean algebra expression is NOT done within the context of chat history. This is both to improve performance and so as to provide more reliable results.

Update State Example:

```
(# Examples of setting the chat state)

("How many children does {user} have?" -> userInfo.children)


(# if the character has children then figure out what the wife's name is)
((userInfo.children > 0)
 ("What is {char}'s wife's name?" -> userInfo.wifeName)
)
```

```
(# always attempt to figure out where the user went to school)
("Where did {user} go to school?" -> userInfo.school)
```

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

Basic 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}
```

Advanced Prompt Template Example:

```
This sentence will always be rendered.

(# this is a comment that will not be rendered)

(("is the following an angry conversation?: {conversation.summary}" && "Is
this a calm personality?: {userInfo.personality}")

This will only be rendered if the condition above is true
The LLM will evaluate the above 'soft code' functions and provide a
boolean response

  ("The current time of {time} is in the evening")

    This nested sentence will only be rendered if both conditions are true

    (# attempt to figure out where the user went to school this will only
be updated if both conditions above are true)
    ("Where did {user} go to school?" -> userInfo.school)

  )


)

This sentence will also always be rendered


(#
This will update the state constantly as the conversation progresses if
the expression is true
Note that userInfo.children will be coerced into a number by the LLM as
best as it is able
)
((userInfo.children > 0)
 "What is {char}'s wife's name?" -> userInfo.wifeName
)
```

AIPL Syntax (informal description)

AIPL borrows parts of its syntax from LISP. It is important to note that while it is 'inspired' by LISP's simple syntax structure, it is not presently a LISP in terms of interpretation.

AIPL allows for both 'natural language' and more structured syntax to live together in the same text. It does this by using the opening and closing `()` symbols to denote where the 'code' parts live.

Template Text

As described above, AIPL has the concept of 'template text' sections that are interpreted as text that will be emitted by the evaluator to be used for things like LLM system prompting. Any character of natural language is permittted with the following exceptions:

- characters between `{}` or `{{}}` are interpreted as state variables
- unescaped `()` characters are interpreted as AIPL 'code sections'

Comments

`(# some comment here)` represent a comment. Note that the immediate `#` after the `(` is what signifies to the parser that a comment is forthcoming.

Escapes

`\(` and `\)` escape the two critical aspects of the syntax if they need to be used in the 'template text' section of AIPL. It is expected that future escapes will follow a similar pattern if needed.

Identifiers

State variables typically will follow the form `<namespace>.<key>`

Operators

In general the traditional C-like boolean operators behave as expected. However some operators have a 'looser' interpretation such as AND and OR.

- `&&` same as `&`
- `||` same as `|`
- `>`
- `<`
- `=` same as `==`
- `!`
- `!=` same as `!==`

Special Operators

- `->` used for assignment where the identifier on the right is assigned-to using the expression on the left. NOTE this is 'backwards' from C-style assignment.

Quoted String Literals

Both double and single quotes are supported.

Use double-quotes like "The user's name is: {user.name}"

Use single quotes when one needs more control over how curly-bracket `{}` characters are interpreted when using chat-state template variables

In general the number of single-quotes before and after a quotation indicates the number of curly-brackets to use for the template variable.

Example:

```
'I have a {red.bucket}`

''I have a {{red.bucket}}``

'''I have a {{{red.bucket}}}'''
```

This is especially useful for quoting other data-types like JSON where `{}` also have meaning.

JSON paired with double-single-quote example:

```
''{"uName":"{{user}}}''
```

NOTE: currently only 3 levels of single-quote are supported, with possibility that more to to come if needed.

Numbers

Integer values of the form `/[0-9]+/` are recognized it is possible that this will be extended to include floating points or more complex number forms later.

Lists

Lists have the form `a,b,c` with optionally-whitespaced comma seperation

Entries

The 'atoms of maps' entries are key-value pairs and have the form `a=b` or `a:b` NOTE that within AIPL using a `:` or a `=` as the seperator of the pair can have different meanings (see URL functions)

URL Functions

The standard URL of the form `http://www.example.com` is interpreted to have special meaninging within AIPL. It is a 'first class citizen'.

There are two forms of 'URL Function' that are recognized for performing both a POST and a GET withing AIPL.

These functions can both assign values to chat-state and output chat-state into the outside world.

GET URL Functions

Using the familiar syntax of a typical URL link, a URL function can be used to assign values to state variables.

`(http://example.com -> example.foo)` will make a GET method HTTP call and return the text value of the result to the state variable `example.foo`. NOTE that in AIPL all state variables are *strings*. This means that regardless of what the Content-Type of the result returned from the URL it will *always* be stored in UTF-8 or similar string form.

The user can also append the standard parameters to URLs like `http://example.com?q=something`. NOTE that currently there is no way to expose chat-state via GET methods. This is likely to change in the future, but for now using the POST form will handle this case.

POST URL Functions

Some liberties have been taken with the standard URL format to make the syntax of 'calling a URL' feel more function-like.

```
(# complex POST webservice with HTTP header and JSON mixed

Note that:
':' is for JSON 'body'
'=' is for HTTP header
'==' is for specials (setting method other than GET/POST)
)
(https://foobar.com(AUTHORIAZTION="bearer super-secret",
method=='PUT',
alpha:"beta",
gamma:"template works {some.value}")
-> foo.baz)
```

Note that JSON is the only data-type supported for POSTs currently, and that the call will set the Content-Type of the 'fetch' to `application/json` and send a correctly formatted JSON string.

The POST form of the URL Function is a standard URL with the extension of a `()` part that accepts entries (see entries above) that will be sued for the header and body section of the HTTP call. (see example)

Note that assignment is 'required' however the language-user is free to ingore the captured results from the URL Function call.

Conditional Expressions

One of the main uses of AIPL is to modify system prompts. This is primarily done via a conditional expression syntax fo the form `(<expression> <body>)`.

If the `<expression>` evaluates to true then the body of the expression is evaluated further.

Note that conditional expressions can be nested within the of a parent conditional expression to the practical limits of memory.

```
((foo.count > 42)
  This will be emitted as text for system prompting, and will not be
emitted if the above condition is false.
)
```