ChatScript

Manuel d'utilisation basique de ChatScript

Revision 9/24/2017 cs7.55

Traduction Mathieu Rigard 02/12/2017 pour Utopia : m.rigard@utopia-french.tech https://utopia-french.tech

A faire : corrections, ancrer les liens.

Présentation

ChatScript (CS) est un langage de script conçu pour traiter l'entrée d'un texte par l'utilisateur et générer une réponse textuelle en retour.

Le chat se déroule en volées, comme le tennis. Le programme traite une ou plusieurs phrases de l'utilisateur et émet une ou plusieurs phrases en réponse.

Couramment, c'est ce qu'on appelle un Chatbot, un programme qui prend les commentaires d'un humain et produit une réponse. Il converse avec vous. Mais j'ai également construit un Chatbot dédié à la lecture de document. Ce qui est traité, dans ce cas là, c'est l'intégralité d'un document.

Quoi qu'il en soit, la sortie générée par le Bot dépend de son programme. Mon programme a été conçu pour trouver la table des matières, la limite de chaque chapitre et l'index à la fin de l'ouvrage.

Donc, plus largement, ChatScript est un système de manipulation du langage naturel, pas uniquement pour la programmation d'un Chatbot.

Ce document traite de la façon d'écrire un script. Vous trouverez une approche plus générale de la façon de concevoir et de créer un Bot dans le document « Écrire un Chatbot », présent dans la documentation.

Texte en entrée et en sortie

ChatScript prend une ou plusieurs phrases entrées par l'utilisateur. Une phrase dans CS signifie nominalement la séquence de mots jusqu'à certains caractères-Token déterminants la fin de la phrase. Par défaut, voici les caractères de fin de phrase :

Le point	Le point d'interrogation	Le point d'exclamation	Le point-virgule	Les deux points	Le tiret
.	?	!	;	:	-

Par exemple : la phrase de utilisateur suivante est divisée en trois phrases:

Quel est ton nom ? Le mien est Alfred ; Je viens de Londres.

On peut aussi désactiver cette façon de déterminer les fins de phrase.

ChatScript émet une ou plusieurs phrases en retour, pour l'utilisateur.

Je m'appelle Harry!
Je suis une démonstration de chatbot, de Californie (USA).

Règles

Fondamentalement, un script est une série de *règles*.

Une règle complète comporte un ***type***, une ***étiquette***, un ***modèle*** et un ***Répondeur*** . Voici une règle simple et complète:

?: VIANDE (vous aimez la viande) Oui j'aime ça

Le ***Type*** de cette règle est **?:** , Ce qui signifie qu'elle réagit uniquement aux questions. ***L'Étiquette*** est VIANDE. Le ***Modèle*** est entre () et cherche à trouver les mots : vous aimez la viande dans cet ordre précis, n'importe où dans l'entrée. Dans cet exemple, à la question *Aimez-vous la viande*. La *réponse* sera Oui j'aime ça.

Le ***Type*** restreint le type d'entrée pour lesquelles la règle s'appliquera. Vous pouvez restreindre les règles aux déclarations, aux questions, ou seulement lorsque le Bot prend le contrôle de la conversation et propose une sentence (c'est alors un *Gambit*). Les ***Types*** sont une lettre suivie par deux points.

L'Étiquette est facultative et sert en particulier à la documentation et au débogage. Elle permet également à la règle d'interagir avec d'autres règles.

Un ***Modèle*** est un ensemble de conditions plus spécifiques qui permettent ou bloquent cette règle, le plus souvent par la correspondance des mots de la phrase entrée, mais parfois en tenant compte de l'historique antérieur de la conversation, de l'heure du jour ou de tout autre élément.

Le ***Répondeur*** est ce que cette règle fait si elle est autorisée à s'exécuter. Étant donné que l'objectif de l'ensemble est de générer une réponse, la sortie la plus simple est simplement la phrase à dire. Les réponses plus complexes peuvent mettre en œuvre l'exécution de conditions, de boucles, l'appel de fonctions etc.

Le système exécute normalement des règles dans un ordre spécifique, les unes après les autres, appliquant les restrictions de type et de modèle, puis génère des réponses destinées à l'utilisateur. Une fois que la réponse est sortie, le système est terminé (sauf si vous souhaitez explicitement faire plus).

Les sujets (Topics)

Les règles sont regroupées dans des collections appelées ***topics***.

Vous dites au système d'exécuter un ***sujet*** (***topic***), et il commence à exécuter des règles sur ce sujet jusqu'à ce qu'il génère une sortie.

Les sujets peuvent invoquer d'autres sujets. La manière exacte de traiter une entrée est contrôlé par un *script de contrôle* qui est lui-même un *sujet* (*topic*). Il appelle les fonctions du moteur et exécute suivant ses conditions d'autres sujets.

Syntaxe du code

Encodage CharSet

ChatScript lit les fichiers texte ASCI et les fichiers **UTF-8.** Il ne lit pas correctement les fichiers UTF-16.

Espaces vides

ChatScript ignore généralement les espaces vides en surnombre. Vous pouvez avoir une pléthore de tabulations, d'espaces et de retours chariots.

Par exemple, les règles:

```
?: (Quand mourrez-vous) je ne sais pas.
```

et:

```
?: (Quand      allez-vous       mourir)
        Je ne sais pas          .
```

sont interprétés de la même manière.

Mais, vous devez utiliser des espaces pour séparer les tokens.

Casse

ChatScript est insensible à la casse pour dans le code du script lui-même. De toute évidence, la casse est importante dans la production littérale.

Et les **mots dans les parenthèses doivent être en minuscules**, à moins qu'ils ne soient les noms propres ou le mot «I» (« Je » en anglais).

REMARQUE: Ne mettez pas un mot minuscule en majuscules dans un ***modèle*** simplement parce qu'il commence une phrase.

commentaires

Le caractère de commentaire est le hachtag #.

Il s'étend jusqu'à la fin de la ligne. Les commentaires ordinaires doivent avoir au moins un espace après eux, pour permettre au système de distinguer les commentaires à partir de constantes numériques comme #NOUN

```
# cette ligne est un commentaire
s: (Je déteste la viande) Donc moi. # un commentaire à la fin d'une réponse
```

Des commentaires spéciaux sur la documentation commencent par #! (développé dans un autre document).

```
#! Je déteste la viande
s: (je déteste la viande) vraiment?
```

Nom des déclarations légales

Un *Topic (sujet)* ou une *fonction* (défini plus loin) ne doit contenir que des caractères alphanumériques, des caractères de soulignement (_), des tirets (-) et des espaces (.). Ils doivent commencer par ~ et continuer avec un caractère alphabétique au départ.

Les variables doivent commencer par $ ou $$ ou $ _, puis continuer avec un caractère alphabétique au départ, puis continuer encore, avec les numéros alphanumériques, les caractères de soulignement ou les traits d'union.

La démo « Hello Word »

Faisons fonctionner le système dans une démo simple.

Windows

Téléchargez et extrayez sous Windows dans un dossier (le mien s'appelle ChatScript), en gardant les fichiers dans leurs dossiers respectifs. Double-cliquez sur le fichier chatscript.exe dans le dossier BINARIES.

Linux

L'exécution sur un système Linux n'est pas très différente. Téléchargez et extrayez dans un répertoire sous Linux, en gardant les fichiers dans leurs dossiers. L'exécutable livré avec le produit est LinuxChatScript64 (version 64 bits) dans le dossier BINARIES. Vous devez modifier les autorisations pour la rendre exécutable. En outre, la version Linux prend par défaut le mode serveur, donc vous devez l'exécuter comme suit:

```
BINARIES / LinuxChatScript64 local
```

Cela ne fonctionnera PAS si vous avez un Linux 32 bits et vous devrez recompiler sur votre machine. Voir Installation et mise à jour du manuel ChatScript.

MacOs

Avec MacOS, le fichier exécutable change:

```
BINARIES / MacChatScript local
```

Le système imprime un tas de statistiques car il charge des données. Après quoi il dit:

```
Enter user name: (Entrez le nom d'utilisateur)
```

Sélectionnez un nom d'utilisateur. C'est arbitraire, mais c'est ainsi que le système vous connaît lorsque vous démarrez plus tard. Le système répondra avec message:

```
Welcome to ChatScript (Bienvenue dans ChatScript)
```

Maintenant saisissez *(à cette date la version de Harry proposée est en Anglais, donc doit être interprété dans cette langue*) : *Quel est votre nom ?* - vous obtenez *mon nom est Harry* si vous avez taper ceci correctement. Vous pouvez également demander *comment était votre enfance ?* et *de quoi avez-vous peur* et *quelle est votre histoire*.

En fait, vous pouvez dire ou demander quoi ce que vous voulez et obtenir une réponse presque raisonnable car Harry discute. S'il ne connaît rien de précis sur votre sujet (ce qui est presque toujours le cas), il peut se bloquer et se laisser aller à la place. Vous pouvez donc discuter de tout.

D'accord. Maintenant, vous avez une idée de ce qui est préinstallé.

Si vous voulez vous perfectionner dans le fonctionnement du Bot Harry et essayer de le changer, lisez le document BOTHARRY. De même si vous voulez construire votre propre robot (en commençant par cloner Harry). Examinons rapidement ce qui se construit.

Aperçu rapide des fichiers de sujets « topic » (.top)

Chatscript est tellement simple qu'un enfant pourrait le maîtriser en quelques secondes. Et si vous y croyez, j'ai plusieurs autres trucs à vous vendre, vous êtes un bon client. Mais comme certaines personnes essaieront de plonger directement, sans lire le reste de la documentation, voici un petit guide de ce que trouverez dans un fichier Topic simple.

Les règles commencent par `t:` ou `?:` ou `u:` ou `s:`

- `s:` signifie que la règle réagit aux *instructions*.
- `?:` signifie que la règle réagit aux *questions*.
- `u:` signifie que la règle réagit à l'union des deux.
- `t:` signifie que la règle offre un *gambit* de sujet lorsque chatbot a le contrôle

Les règles commencent également par `a:` `b:` etc., mais ce sont des *réponses*, pas des règles de niveau supérieur. Elles sont utilisées pour anticiper comment un utilisateur peut répondre à la sortie et donner un retour direct en fonction de cette réponse.

Les règles sont donc classées comme suit:

Type de règle	syntaxe	description

Répondeurs	`s:`
	`?:`
	`u:`
		qui sont des règles qui tentent de réagir à l'entrée de l'utilisateur non provoquée par le Bot. Dans les cas où l'utilisateur directement une question hors contexte par exemple.

Répliques	`a:`
	`b:`
	`c:`
	`d:`
	`e:`
	`f:`
	`g:`
	`h:`
	`i:`
	`j:`
	`k:`
	`l:`
	`m:`
	`n:`
	`o:`
	`p:`
		sont des tentatives pour prédire la réponse immédiate d'un utilisateur à quelque chose que le chatbot dit. Ils ne sont déclenchés que si la saisie survient immédiatement après que la règle qu'ils suivent a été utilisée.

Type de règle	syntaxe	description
gambits	q: r: t:	c'est l'histoire que le chatbot amène sur un sujet ou la conversation dans laquelle le chatbot essaie de diriger l'utilisateur. r: sont des gambits aléatoires (expliqué plus loin)

Les règles présentent généralement leurs ***modèles*** entre parenthèses (sauf les gambits t: règles pour lesquelles un modèle est facultatif). Ceux-ci essaient généralement de trouver des mots ou des séquences spécifiques de mots dans l'entrée de l'utilisateur. Dans la règle:

```
u: (prendre la fuite)
```

le système cherchera si le mot « prendre » peut être trouvé n'importe où dans la phrase, et si oui, s'il est immédiatement suivi du mot « la » puis « fuite ». Par contre, à chaque fois que vous voyez [...] cela signifie qu'il cherchera les mots séparément.
Ainsi :

```
u: ([effrayant peur])
```

signifie que le système doit trouver n'importe où dans la phrase un des mots *effrayant* ou *peur*. Et la *peur* peut être dans l'une de ses formes connexes (c'est la notion de peur qui est cherchée) : *peur, apeuré, terrifié*.

Dans les fichiers « sample », vous verrez des mots ordinaires et des ~mots. Les mots précédés du tiret « Tilde » se réfèrent à un groupe de mots conceptuel, une liste de mots qui se rapproche du ~mot. Par exemple.,

~aime représente un ensemble de mots qui signifient aimer quelque chose ou quelqu'un.

~animaux signifie une importante liste de noms d'animaux.

Ce sont des raccourcis intéressant à utiliser avec la recherche []. Au lieu d'avoir à écrire [éléphant de tigre léopard alligator crocodile lion ...] dans chacune des règles, lorsque le concept est enregistré on peut simplement dire ~animaux.

Les règles sont toujours regroupées dans une rubique ou « Topic », comme le sujet: ~enfance peut former un Topic. Les rubriques contiennent une liste de mots clés pertinents qui suivent directement le nom de la rubrique (du Topic). Passée cette première ligne, les règles peuvent être indiquées dans n'importe quel ordre. Les gambits, ces lignes qui commencent par un t:, offrent une histoire ou un flux de conversation prévu.

Si vous demandez des informations dans une conversation, vous devez équilibrer la balance en fournissant des informations. Par exemple, si vous

demandez ce que quelqu'un apprécie comme passe-temps, il sera attendu après la réponse une réponse à la même question. Comme dans:

```
topic: école [école université apprendre]

t: Où allez-vous à l'école ?
t: Je vais à Harvard.
t: Dans quelle filière ?
t: J'étudie la finances.
```

Et votre sujet doit être prêt à traiter des questions arbitraires liées à l'école de l'utilisateur.

S'il dit : « je vais à Yale. Quelle était la mascotte de votre école ? », vous aimeriez avoir ajouté des réponses à la question de la mascotte avant de passer à l'anticipation volontaire (via les t:) annonçant que vous êtes allé à Harvard.

C'est ce que le Topic simple sur l'enfance tente de faire. Poser des questions *Gambit*, répondre avec des remarques appropriées à leur réponses, proposer des réponses concernant le chatbot à ces même *Gambits*, passez à la suite et suivez quelques questions simples posées sur le sujet dans le Topic.

Les commentaires commencent par *#*. Dans le fichier, le commentaire *#* est seul, il s'agit d'un commentaire ordinaire.

Commentaires spéciaux *#!* donne un exemple d'entrée par l'utilisateur auquel la règle qui suit immédiatement doit correspondre et gérer. Ceci indique à la fois quelles entrées devrait correspondre à la règle ci-dessous et permet au moteur de la tester automatiquement.

Le commentaire spécial donne un seul exemple d'entrée correspondante, pas toutes les entrées possibles pouvant correspondre. Cela vous aide à comprendre ce à quoi **un Répondeur** ou **une Réplique** sont censé réagir. Il n'a aucun impact sur un utilisateur en conversation, cela aide simplement la lecture du code.

Quels fichiers avons-nous ?

Le moteur ChatScript peut exécuter plusieurs bots à la fois (voir le manuel ChatScript Multiple Bots), chacun avec une personnalité unique. Ainsi, un utilisateur peut se connecter et parler avec une personnalité spécifique tandis qu'un autre utilisateur se connecte et parle avec un autre (ou le même).

Répertoires	**Fichiers**
BINARIES	Fichiers exécutables. Tous les exécutables sont dans le dossier BINARIES. Le système suppose qu'ils sont lancés à partir de là (est le répertoire de travail actuel) et modifie le répertoire d'un niveau pour accéder à tous les autres

	dossiers.
USERS	Fichiers d'historique. La conversation de chaque utilisateur est suivie par le système et conservée dans des fichiers du répertoire USERS. Un utilisateur peut revenir au même Chat des jours plus tard, et le système sait ce qui s'est passé dans les conversations précédentes et que c'est le début d'une nouvelle conversation. Le système conserve également un fichier journal par utilisateur enregistrant ses conversations pour l'auteur (CS n'utilise pas ce fichier, il l'enregistre simplement) et un fichier topic pour chaque duo utilisateur/chatbot, où il stocke l'état actuel de la conversation pour le moteur. Si le script enregistre des faits sur l'utilisateur pendant la conversation, ceux-ci sont également stockés dans le fichier topic_*.txt.
DICT	Fichiers du dictionnaire. Le dossier DICT est le dictionnaire sous-jacent du système. Vous ne le modifierez probablement pas. Il a un sous-dossier ENGLISH qui est un dictionnaire complet qu'il utilise par défaut. Si vous essayez d'exécuter ChatScript sur un appareil mobile, vous souhaiterez probablement copier sur le contenu de BASIC dans le dossier ENGLISH, pour utiliser un dictionnaire plus petit.
LIVEDATA	Fichiers d’Extension du Dictionnaire. Le dossier LIVEDATA contient des extensions du dictionnaire et du système d'exécution que vous pouvez modifier en tant qu'utilisateur avancé.
RAWDATA	Fichiers de connaissances. Le dossier RAWDATA est l'endroit où sont conservées les données brutes pour prendre en charge le Chat (bien que vous puissiez la garder n'importe où car changer n'est pas compilé dans le moteur). Il existe des dossiers pour HARRY, ONTOLOGIE, WORLDDATA, STOCKPILE, NLTK, POSTGRES et QUIBBLES. Normalement, lorsque vous définissez un chatbot, vous ajoutez un dossier avec le nom de votre chatbot. De cette façon, vous pouvez simplement intégrer les mises à jour du système principal chaque fois qu'une nouvelle version est créée.
TOPIC	Ces données sont exécutées dans le script "compilateur" et la sortie est stockée dans le répertoire TOPIC, qui contient vos données de script compilées.
VERIFY	Si votre script contient des données de vérification incorporées (commentaires commençants par #!), permettant au système de tester si vos modèles font réellement ce que vous voulez qu’ils fassent, ces données sont stockées dans le répertoire VERIFY après la compilation.
SRC	Fichiers sources. SRC est une source pour reconstruire le moteur. Il a un fichier dictionarySystem.h qui est lu pendant le chargement pour définir les constantes du moteur qui peuvent être utilisées dans les scripts.
WIKI	Fichiers de documentation. Ce document et d'autres peuvent

	être trouvés dans WIKI.
LOGS	Si ChatScript détecte les bogues pendant l'exécution, il les stocke dans bugs.txt dans ce dossier. Un serveur stockera également son journal ici.
Compilation Files	Les dossiers LINUX, MAC et VS2010 sont destinés à reconstruire le moteur exécutable. LOEBNERVS2010 construit une version du concours Loebner.
Top Level Files	Outre chatscript.exe, les fichiers de niveau supérieur suivants existent: authorizedIP.txt - en tant que serveur, cela permet à certains utilisateurs d'entrer des commandes changes.txt - liste des changements entre versions version.txt - la version actuelle talk.vbs - script pour activer la sortie vocale dans Windows
REGRESS	Les fichiers qui peuvent faire des tests de régression de différents types sont ici.
Server Batch Files	Fichiers de serveurs pour Windows qui peuvent exécuter le moteur en tant que serveur local ou client local.
Third Party Tools	Les outils de tiers font référence à certains outils tierces de ChatScript.
TMP	Les fichiers transitoires TMP utilisés pour supporter le débogage sont conservés ici.

Les TOPICS simples

Dans un thème, une rubrique (TOPIC), le système n'exécute pas toutes les règles. Vous ne choisissez pas non plus une règle à exécuter. Au lieu de cela, vous (l'auteur) organisez des collections de règles dans des sujets et le système choisit le sujet à exécuter à tout moment. Il exécute à son tour ses règles.

Voici un exemple de déclaration de Topic simple:

```
topic: ~MORT [corps mortel mort mort corps]

t: je ne veux pas mourir
?: (Quand allez-vous mourir) je ne sais pas.
```

Le Topic déclare son nom, ses mots clés, puis ses règles. Il se termine par la fin du fichier ou une nouvelle déclaration de niveau supérieur (qui comprend : `topic:`, `concept:`, `table:`, `tablemacro:`, `outputmacro:`, `patternmacro:`, `dualmacro:`, `bot:`, `data:`, `canon:`, `query:`, `plan:`, `describe:`, et `replace:`).

Le nom du Topic doit commencer par un ~, un caractère alphabétique, puis être un nom légal standard (ne contient que des caractères alphanumériques,

des caractères de soulignement, des traits d'union et des points).

Mots clés

Les mots clés permettent au système de considérer ce Topic en fonction des correspondances avec l'entrée de l'utilisateur.

Si l'entrée de l'utilisateur n'a pas de mots clés en commun avec le sujet, normalement, le sujet et toutes ses règles ne seront pas prises en considération.

Les mots-clés issus d’une phrase de l'utilisateur déclenchent le Topic correspondant le plus proche (basé sur le nombre et la longueur des mots-clés) pour le tester.

Si le moteur peut trouver une réponse correspondant, alors il se déplace vers ce sujet et répond. Sinon, le moteur essaie d'autres sujets correspondants. Finalement, si il ne peut pas trouver de réponse approprié, il peut revenir au sujet le plus pertinent et laissez simplement un Gambit (la règle commençant par `t:`).

Peu importe si le nouveau Topic a des réponses qui chevauchent les répliques d'un autre Topic (les deux pourraient correspondre à la requête). Vous pouvez avoir un sujet sur ~funéraire et une autre sur ~mort. Une phrase d'entrée que je ne croit pas en la mort pourrait vous acheminer vers l'un ou l'autre sujet, mais c'est une bonne chose.

Si vous êtes au milieu d'un Topic et que l'utilisateur dit quelque chose qui est hors sujet, le système cherchera un Topic qui correspond aux mots-clés de l'entrée et verra si un sujet correspondant a une réponse pour réagir à l'entrée. Dans l'affirmative, le système passera à ce Topic.

Le nouveau Topic devient le Topic actif et l'ancien Topic devient « en attente », mis dans la liste des Topics susceptibles de revenir lorsque le nouveau sujet sera épuisé. Ce comportement est le comportement par défaut et vous pouvez créer d'autres comportements en créant un script de contrôle pour le faire (avancé).

Les Topics facilitent le regroupement logique des règles. Les mots-clés d’un Topic signifient que l'auteur peut développer un domaine de conversation indépendant sans tenir compte des Topics préexistant et le système le trouvera automatiquement et l'invoquera selon le cas.

Les Topics ne doivent pas obligatoirement avoir de mots-clés. Mais pour qu’une règle dans un Topic puisse être appelée, soit le Topic a des mots-clés qui peuvent être trouvés dans la règle elle-même, SOIT, le Topic est déjà le Topic courant (une autre règle avait des mots-clés dans son modèle qui étaient également mots clés du Topic alors vous êtes déjà dans le Topic) SOIT enfin, le Topic a été directement appelé par le script de contrôle ou un autre sujet.

Règles des Gambits

En plus des **Répondeurs** réagissant aux entrées des utilisateurs (`s:, ?:, u:`) un Topic peut avoir des Gambits à proposer (`t:`).

Les Gambits créent une histoire cohérente sur le sujet si le Bot maîtrise la conversation.

Pourtant, si l'utilisateur pose des questions, le système peut utiliser un **Répondeur** pour répondre directement ou réutiliser un Gambit qu'il avait prévu de proposer de toutes façons. C'est entièrement à vous de décider de l'ordre des **Répondeurs** et des **Gambits**. Vous pouvez séparer `s:` de `?:` de `u:` ou les mélangez.

Vous pouvez avoir des **Répondeurs** mélangées à des **Gambits**.

Qu'est-ce que cela signifie pour le chatbot d'être "en contrôle" de la conversation?

Lorsque l'utilisateur tape une déclaration ou une question, le système verra si il a une **Réplique** ou un **Répondeur** qui correspond directement à l'entrée. Si c'est le cas, il répondra avec. Cette réponse « forcée » signifie que l'humain maîtrise la conversation. Si, cependant, le chatbot ne peut pas trouver une réponse pour l'entrée, alors le chatbot est libre de dire tout ce qu'il veut. Alors il est en contrôle.

Peut-être va-t-il émettre un Gambit du Topic dans lequel il était précédement. Peut-être partira-t-il sur un Gambit d'un Topic étroitement lié à l'entrée de l'utilisateur. Peut-être essaiera-t-il de diriger la conversation vers un nouveau sujet spécifique, un Topic différent. Tout ce qui est contrôlé par la logique du script de contrôle (dont il existe un code par défaut fourni avec ChatScript mais que vous pouvez modifier).

Le Gambit typique n'a pas **d'Étiquette** ni de **Modèle**. Il pourrait en avoir. Il se compose généralement uniquement de son type de règle (`t:`) et des données de sortie.

L'ordre d'exécution

Un sujet est exécuté en mode **Gambit** (c'est-à-dire t: et la ligne qui suit) ou en mode **Répondeur** (avec `s:, ?:` et `u:` et leurs lignes). Les règles sont placées dans l'ordre où vous voulez qu'elles soient essayées.

Dans le cas des Gambits, l'ordre raconte une histoire.

Pour les **Répondeurs**, les règles sont ordinairement classées des plus

spécifiques aux moins spécifiques, éventuellement groupées par thème. Donc, une **Répondeur** essayant d'enregistrer la couleur de vos cheveux viendrait avant celle qui réagirait simplement à toute référence à vos cheveux.

Par défaut, le système évite de se répéter, marquant comme règles utilisées celle qui ont déjà généré une émission par le Chatbot. C'est ainsi que l'histoire proposée par un Topic peut être racontée. Le premier Gambit est exprimé, marqué comme lu, puis ensuite, lorsque sortira le second gambit il sera à nouveau marqué comme lu, et ainsi de suite.

De même, un **Répondeur** qui réagit à une entrée délivrera son message, puis s'effacera. Si l'utilisateur répète son apport, cette règle ne sera plus utilisée, et une autre règle devra répondre à sa place.

Le fichier `RAWDATA/skeleton.top` comporte un tas de Topics déjà prédéfinis avec des mots-clés, mais pas de Répondeurs ou de Gambits.

Si vous avez rempli certains de ces Topics avec des règles et avez enregistré le lien dans le fichier `RAWDATA/filesHarry.txt` et reconstruit les données (`:build`), vous avez un Chatbot !

Les Répliques

Si vous vous attendez à ce que l'utilisateur réponde d'une manière particulière à votre sollicitation, vous pouvez modifier le script pour examiner sa prochaine entrée et voir si elle correspond. Quand cela fonctionne, votre Chatbot semble comprendre l'utilisateur. Cela s'appelle des **Répliques** et toutes les règles peuvent en posséder.

```
s: (J'aime les épinards) Êtes-vous un fan des dessins animés de Popeye?
    a: (oui) Je les ai regardé enfant. Avez-vous aimé Olive Oyl?
        b: (non) Moi non plus. Elle était trop maigre.
        b: (oui) Vous aimez probablement les femmes minces.
    a: (non) Quelles dessins animés regardez-vous?
        b: (aucun) Vous menez une vie monacale.
        b: (Mickey Mouse) La Star de Disney.
```

Les **Répliques** utilisent de `a:` à `q:` pour indiquer la profondeur de la séquence.

Toutes les **Répliques** de même niveau ont la même lettre et sont des alternatives qui seront testées dans l'ordre. Alors, après que le Chatbot ait demandé `Êtes-vous un fan`... il va tester la prochaine entrée pour oui, puis non. Dès qu'il trouve une réplique en correspondance, il l'exécutera et se terminera. S'il ne trouve aucun, alors le système passe à son comportement normal.

Les **Répliques** peuvent avoir des **Répliques**, comme indiqué ci-dessus. Une présentation comme ci-dessus est correcte, elle rend visiblement évident l'agencement des répliques dans votre script.

Remarque

Techniquement, on utilisera pas `oui` et `non` réellement comme cela. Ils seront considérés comme spéciaux et traités comme des interjections (comme toutes les autres synonymes ayant de nombreuses façon d'exprimer la même chose). Pour que l'exemple ci-dessus fonctionne réellement dans le moteur, vous devriez utiliser ~oui et ~non. Mais vous apprenez les concepts et les interjections plus tard.

Étiquettes de règles

Toutes les règles (Répondeurs, Gambits, Répliques) peuvent avoir des **Étiquette** liées. Les étiquettes peuvent être utilisées de façon très variées. D'autres règles peuvent utiliser des fonctions ciblant une règle étiquetée particulière. Vous pouvez utiliser les capacités de débogage pour tester une règle et ainsi vous pouvez tracer une règle pour la vérifier. Et vous obtenez alors une sorte de documentation vous indiquant votre règle.

Une **Étiquette** est un mot unique placé entre le type de règle et son **Modèle**. Si la règle est un Gambit, vous devez forcément ajouter un **Modèle**, même si ce n'est qu'une parenthèse vides.

```
t: MES_YEUX () Mes yeux sont bleus

?: COULEURDESYEUX (couleur * yeux) J'ai des yeux bleus

u: LUNETTES ([contacts lunettes]) Je ne porte pas de lunettes, mais j'ai des lentilles de contacts.

?: AVEUGLE (vous * aveugle) Je ne suis pas aveugle.

?: DALTONIEN (vous * [daltonien "pas voir le vert")) Je ne suis pas daltonien.
```

Note, ChatScript est une langue orientée Token (les jetons sont généralement une suite de caractères sans espaces). Vous devez placer un ou plusieurs espaces après les `:` et avant l'étiquette. AUTREMENT `?:COULEURDESYEUX` est un jeton unique et non la déclaration d'une règle (`?:`) et d'une étiquette.

Le fichier `simpletopic.top` comporte un exemple appelé `~Childhood` d'une complexité normale (qui peut être compris après avoir lu « advanced output »).

Les modèles simples

Comme indiqué précédemment, une règle ne peut pas déclencher à moins que son modèle ne corresponde, et un modèle, dans une règle, se trouve placé entre parenthèses (et ses éléments y sont présentés dans un ordre précis formant une séquence).

Les écrire un modèle est demande précision et doigté. Si votre modèle est trop spécifique, il manquera toutes sortes de possibilités de répondre à des significations similaires. Si votre modèle est

```
?: (quand allez-vous rentrer chez vous) Je rentre chez moi demain
```

et l'entrée est « `quand allez-vous à la maison` », le Bot ne réagira pas. Mais si votre modèle est trop large, le robot répond à des significations complètement erronées. Si votre modèle est

```
s: (maison) Je rentre chez moi demain.
```

alors il réagit à « `connais-tu les elfes de maison ?` » de manière inappropriée.

La séquence ( )

J'ai dit que les parenthèses étaient comprises par le système dans l'ordre de la séquence proposée, n'importe où dans l'entrée. Ainsi

```
s: (je t'aime) M’aimez-vous vraiment ?
```

correspondra à Comme je t'aime ! et `je t'aime et j’aime toutes les femmes` et `tout le monde sait que je t'aime`. Vous pouvez même mettre des parenthèses dans des parenthèses, non que ça ait une quelconque utilité.

```
s: (je ( t'aime)) M’aimez-vous vraiment ?
```

Ce modèle est le même que le précédent sans les parenthèses imbriquées. Alors que les parenthèses extérieures font commencer leur premier élément correspondant à n'importe quelle partie de la phrase, la seconde parenthèse en hérite et les éléments contenus dans cette dernière sont sélectionnés consécutivement. Ainsi, le contenu de la seconde parenthèse dois venir consécutivement dans la phrase à celui de la première pour que la règle soit valide.

Une autre façon de demander une séquence est de proposer des guillemets de part et d’autre.

```
s: ("Je t'aime") M’aimez-vous vraiment ?
```

Il y a deux raisons d'utiliser le double guillemets. Tout d'abord, si vous essayez trouver une suite de mot qui exprime une idée. Par exemple, comme mot-clé d'un sujet, vous pouvez faire:

```
topic: ~mort ["être mort" "passer l’arme à gauche"]
```

Deuxièmement, lorsque vous essayez de spécifier une idée pour laquelle vous n'êtes pas certain de la façon elle apparaîtra ni s'il s'agit d'un mot ou d'une séquence de mots.

Une mise en Token impliquant une ponctuation peut être délicate. Par exemple, le mot `Bob's` est en fait « tokenisé » comme deux mots: Bob et s. Et dans ce qu’on peut trouver sur le Net, `New_Year's_Eve` est un mot unique.

Vous ne le savez peut-être pas, mais chaque fois que vous pensez à quelque chose sous forme de mots multiples, vous êtes sûr de ne pas vous tromper en le mettant entre guillemets (ex : «Nouvel An») et en laissant ChatScript gérer la façon dont il est stocké en interne. Ceci est particulièrement vrai pour les noms de personnes, les titres de livres et d'autres noms propres à plusieurs mots.

Mettez la ponctuation dans les guillemets pour être sûrs. Le modèle correspondant à une séquence est limité à 5 mots à la suite et les reliera à la fois aux formes originales et canoniques.

Limiter les phrases avec < et >

Parfois, pour avoir une signification appropriée dans le modèle, vous devez effectivement savoir où une entrée commence et se termine. Par exemple:

```
u: (Qu'est-ce qu'un éléphant) Un éléphant est un pachyderme.
```

Fonctionne avec `Dites-moi qu'est-ce qu’un éléphant` et `qu'est-ce qu’un éléphant` et `qu'est-ce qu'un éléphant fait dans la salle`. Ce dernier est mal adapté.

Le > correspond à la fin de la phrase. Cela permet de gérer correctement les phrases ci-dessus comme suit:

```
u: (Qu'est-ce qu'un éléphant > ) Un éléphant est un pachyderme.
```

Le < retourne au début de la phrase pour trouver une concordance avec ce qui lui est lié (ici : `roses)`. Ainsi

```
u: (roses < j'aime) J'aime les roses aussi.
```

Fonctionne avec `J'aime les roses` parce qu'il trouve `roses` n'importe où dans la phrase, puis le < réinitialise la position de la correspondance au début de la phrase, puis il trouve le `j'aime` au début. Bien sûr, cela ne fonctionne pas avec `Vous savez que j'aime les roses` parce que `j'aime` n’est pas au début de la phrase.

Le Joker (ou Troncature) simple indéfini *

Le caractère générique * signifie 0 ou plus de mots en séquence successive. Il peut être utilisé pour élargir un modèle :

```
?: (quand * vous * à la maison) je rentre à la maison demain
```

Ce modèle répond à `Quand allez-vous aller à la maison` et `Quand Roger est avec vous, y a-t-il quelqu'un à la maison ?`

Joker précis *n

Comme vous le remarquerez peut-être, les Joker indéfinies peuvent permettre à toutes sortes de méfaits de se faufiler dans une correspondance. Une manière plus précise de gérer le Joker consiste indiquer exactement combien de mots peuvent être couverts par l'astérisque. Le * suivi d'un nombre représente combien de mots il couvre.

```
?: (quand *1 vous *1 à la maison) Je suis rentré à la maison hier
```

Fonctionne avec `Quand êtes-vous rentré à la maison` mais n'acceptera pas de grandes variations comme `quand Roger est avec vous à la maison` ni `quand vous rentrez à la maison`, car le le premier * 1 n'a pas pu s'exprimer.

Joker à restriction de portée *~n

La manière habituelle de gérer les excès des caractères génériques précédents consiste à utiliser un caractère générique restreint. Il s'agit d'un * suivi d'un ~ et d'un nombre, comme *~3. Cela signifie à partir de 0 jusqu'à ce numéro, ou approximativement ce nombre.

Un choix courant est *~2. Cela laisse place à certains mots de remplissage (comme un déterminant et un adjectif ou peut-être un type d'adverbe), sans les exiger ou laisser la phrase s'étendre.

```
?: (vous *~2 aller *~2 à la maison) Je vais souvent dans cette maison.
```

Cela répond également à `Vous pouvez aller à la maison` et à `vous pouvez toujours aller à la maison`.

Les marqueurs non désordonnés << >>

Souvent, vous voudrez faire correspondre plusieurs mots-clés, mais vous ne les voudrez pas dans n'importe quel ordre. Par exemple, la phrase `j'aime les oiseaux` ressemble beaucoup à `les oiseaux c'est ce que j'aime`, mais le sujet et le complément ont changé de place. Une façon un peu fastidieuse de les faire correspondre dans l'ordre est :

```
s: (~Je <* aime <* oiseaux) J'aime les oiseaux aussi.
```

Cela fonctionne en remontant au début de la phrase et en permettant à n'importe quel nombre de mots de faire correspondre un caractère générique jusqu'à ce que le prochain mot-clé soit trouvé. C'est moche. La manière la plus propre est d'utiliser les marqueurs non désordonnés.

```
s: (<< J'aime les oiseaux >>) J'aime les oiseaux aussi.
```

Puisque les mots peuvent être trouvés dans n'importe quel ordre, cela réinitialise le mécanisme de numérisation à la condition initiale de départ, qui est toujours <* signifiant que vous pouvez associer l'élément suivant n'importe où dans la phrase.

La position est remise librement au début suivant la séquence << >>, donc, si vous aviez :

```
u: (J'aime << vraiment >> photos)
```

et une saisie comme `Des photos que j'aime vraiment`, fonctionnerait parce que le système trouvera bien `j'aime` n'importe où dans la phrase, puis réinitialisera la position librement pour recommencer et trouvera `des photos` autre part ailleurs dans la phrase.

Les choix []

Vous pouvez combiner des mots différents à la même position dans la phrase en plaçant ces choix entre crochet.

```
?: (vous [nagez pêchez skiez]) Oui ça m'arrive.
```

Cela fonctionnera avec `Vous nagez` et `Est-ce que vous pêchez ?` Comme avec `Vous skiez ?` Les choix peuvent être des alternatives importantes ou ils peuvent être des synonymes.

```
?: (vous [mangez ingérez "vous en mettez plein la lampe" (êtes le prédateur naturel)] *~2 viande) J'adore la viande
```

Notez que les éléments d'un choix peuvent être des séquences de mots soit entre guillemets soit entre crochets.

Les ~Concepts

Les choix sont utiles pour les synonymes, mais vous devez les répéter à chaque fois dans chaque règle. Ainsi, déclarer une liste de choix dans un seul endroit et de les utiliser partout ailleurs devient pratique. C'est ce que permettent les **Concepts**. Il sont très important dans les modèles qui font intervenir le sens plus que le mot lui-même.

```
concept: ~mange [manger ingérer bouffe "met plein la lampe"]
```

Contrairement aux choix, un concept ne peut pas utiliser les parenthèses pour

contenir une séquence de mots, bien qu'il puisse utiliser les expressions entre guillemets.

Un concept est une déclaration de niveau supérieur consistant en un nom commençant par ~ et qui ne comporte que des caractères alphanumériques et des caractères de soulignement. Un concept comporte une liste de mots qui le définit. Vous pouvez utiliser le nom de l'ensemble dans n'importe quelle catégorie de mot-clé ou de thème à la place d'un mot.

```
s: (je ~mange de la viande) Vraiment ? Moi je suis végétalien.
```

Pensez au caractère ~ comme s’il signifiait approximativement. Les noms de **Topic** sont également des noms de concepts, les mots-clés les accompagnant sont leurs **Choix**.

ChatScript peut représenter des synonymes de mots comme ci-dessus ou des mots affiliés comme ci-dessous.

```
concept: ~baseball [arbitre batte base balle]

# une déclaration de Topic

s: (~baseball) Je ne suis fan des sports américains comme le baseball.
```

Un concept peut aussi être un ordre naturel de mots qu'un script avancé peut utiliser. Le concept commandé ci-dessous montre la question de la « main » au poker.

concept: ~pokerhand [quinte royale quinte flush carré de rois full]

Le modèle :

```
?: (qui * mieux * ~pokerhand * ou * ~pokerhand) ...
```

détecte des questions comme qu’est-ce qui est mieux, une quinte flush ou une quinte royale ? et le système a des fonctions qui peuvent exploiter l’ordonnancement du concept pour fournir une réponse correcte.

Vous pouvez encapsuler des concepts dans d’autres concepts, donc il est correct de mettre :

```
concept: ~nourriture [ ~viande ~dessert lasagne ~légumes ~fruit ]
```

L'héritage hiérarchique est important dans la généralisation des modèles. Les concepts peuvent être utilisés pour créer des ontologies complètes de verbes, de noms, d'adjectifs et d'adverbes, permettant de faire correspondre les significations générales ou idiomatiques.

Le système comme il se présente est déjà programmé, il suffit de l'activer. Si vous exécutez la commande :build 0 sur le chatbot, vous construirez l'ontologie sous-jacente et la connaissance qu’à du monde le système. A vous d’explorer les dictionnaires et ensembles conceptuels existants.

En plus des ensembles fixes (plus de 1600), le système définit automatiquement des séries d'ensembles basés sur un dictionnaire. Ceux-ci comprennent des éléments de la phrase comme ~nom ainsi que des concepts généraux comme ~nombre.

Pour une liste complète, voir le ChatScript System Variables and Engine-defined Concepts manual.

Capitalisation

Quelle que soit la façon dont vous tapez votre saisie, avec ou sans majuscules, le système tentera de comprendre le mot que vous voulez dire.

Si vous écrivez : `Puis-je revenir en arrière`, vous ne voulez pas dire que `Puis-je` est un nom propre. Vous voulez dire, `puis-je revenir en arrière`, et c'est ce que le système comprend.

Cela signifie que lorsque vous écrivez des **Modèles**, vous ne devez pas mettre de majuscules au premier mot de phrases. Vous mettrez des majuscules que pour les noms propres. Donc, ce modèle:

```
u: (Puis-je revenir en arrière)
```

est faux et ne correspondra à rien et devrait être

```
u: (puis-je revenir en arrière)
```

Noms propres

Vous devez toujours mettre les noms propres à plusieurs mots entre guillemets, en particulier ceux avec ponctuation intégrée. Vous voulez que CS sache que toute la phrase est considérée comme une entité unique. Alors

```
u: ( "Dr. Watson" )
```

```
u: ( "The Beatles" )
```

Interjections, "actes de discours" et ensembles de concepts

Un certains nombre de mots et de phrases sont interprétées différemment selon qu'ils se trouvent au début où à une autre position dans la phrase. Par exemple : Salut, ça va et c'est pour lui une chance de salut sont différents. De même, C'est sûr et Crois-tu vraiment que c'est sûr ne veulent pas dire la même chose.

Les mots qui ont un sens différent au début d'une phrase sont communément appelés interjections.

Dans ChatScript, ceux-ci sont définis par le fichier interjections.txt (pour l'anglais, voir interjections.txt).

En outre, ce fichier enrichit le concept avec des *actes de discours*, des phrases qui sont elles-même des sortes d'interjections. Toutes les interjections et ces actes de discours correspondent à des ensembles de concepts, qui seront traités par le moteur en lieu et place des entrées réelles de l'utilisateur.

Par exemple, *oui* et *ok* et *bien sûr*, tous sont traités grâce au fichier d'interjections comme signifiant l'acte de discours d'être d'accord. Donc, vous ne verrez pas *oui*, *j'y vais* sortir du moteur. Le fichier d'interjections fera bifurquer (« remapper ») oui par ~yes, en le séparant de la phrase originale, suivie de j'y vais comme nouvelle phrase.

Ces interjections génériques (qui sont ouvertes au contrôle de l'auteur via interjections.txt) répertoriées dans le manuel ChatScript System Variables and Engine-defined Concepts manual.

Parce que toutes les interjections de début de phrase sont ainsi interrompues dans leur propre phrase, ce type de modèle ne fonctionnera pas:

```
u: (~yes _*)
```

Vous ne pouvez pas capturer le reste de la phrase ici, car cela fera partie de la phrase suivante pour le moteur. Cela signifie que les interjections agissent un peu différemment des autres concepts. Si vous utilisez un mot dans un modèle qui peut être remappé lors de la saisie, le compilateur de script émettra un avertissement. Probablement vous devriez utiliser le nom remappé à la place.

Canonisation

Le système vous aide en généralisant vos **Modèles**. Il sélectionne à la fois le mot original et sa forme canonique si le mot de votre **Modèle** est sous sa forme canonique. Il vérifie également les formes minuscules et majuscules de vos mots. Pour les noms, la forme canonique est au singulier. Donc, si votre **Modèle** est:

```
?: (chien) J'ai un chat
```

Cela répondra également à `moi, j'aime les chiens` et à `j'ai un chien`. Alors que le motif

```
?: (chiens) J'ai un chat
```

ne répondra qu'à `j'aime les chiens` mais pas à `j'ai un chien`.

Pour les verbes, la forme canonique est à l'infinitif. Si votre modèle est:

```
?: (avoir *~2 raison) Oui.
```

Répondra de la même façon à `A t il raison ?` comme à `Ont ils raison ?` et `ayons tous raison`. (les passages en rouge sont à revoir ou corriger).

- Les suffixes possessifs `mon ma` sont traités comme `mon`.
- Les adjectifs et les adverbes reviennent à leur forme de base.
- Les déterminants `un le la ce celui-là` deviennent `un`.
- Les numéros en toutes lettres comme `deux mille vingt-et-un` sont transcrits en format numérique.
- Les nombres à virgules sont traduits en nombres entiers s'ils correspondent exactement à la valeur, tandis que les valeurs de devise deviennent des nombres à virgules.
- Les pronoms personnels comme `moi, mon moi-même`, passent à la forme du sujet `je`, alors `qui, qui que soit, celui qui, celui là, quelqu'un` deviennent `quelqu'un` et `quoi, quand`, deviennent `quand` et `que qui-que-ce-soit` devient `lequel.`

Le fichier canonical.txt dans LIVEDATA contrôle la majeure partie de ces derniers.

Si le système voit *&* dans l'entrée, il le modifie en `et`. Cela change aussi _ en `'_`.

Dans ChatScript, le simple concept ci-dessous accepte tous les temps et toutes les conjugaisons des verbes listés :

```
concept: ~être [ être ressemble paraît semble ]
```

Si vous mettez une apostrophe devant un mot ou utilisez des mots qui ne sont pas sous leur forme canonique, le système se limitera à ce que vous avez utilisé dans le Modèle:

`u: ( tu ‘aime bien )` Cela correspond `tu aime bien`, mais pas à `tu aimais bien`.

`S: ( tu est )` Est lu comme `Tu es` mais pas comme `tu étais` ni `tu n’est pas`

Il en va de même pour les concepts:

```
u: ( '~adverbes_etendus )
```

Exclamation ! Et double exclamation !!

On peut également rechercher l’absence d’un mot à un certain endroit de la phrase. L’utilisation de `!` signifie que ce qui est cherché à partir ce cet emplacement ne doit pas être trouvé dans la suite de la phrase pour que la règle s’exécute. Lorsqu'il est placé au début du Modèle, on ne dois le trouver nulle part ailleurs dans la phrase:

```
u: (![ pas jamais rarement ] je ~ingérer * ~viande) Vous mangez de la viande.

u: (! ~négation je * ~aime * ~viande ) Vous aimez la viande.
```

Alors que ! vérifie tout le reste de la phrase, !! vérifie uniquement le prochain mot après l'endroit où vous vous trouvez. Alors

```
u: (test !!ça) Je gagne
```

fonctionnera avec la saisie `teste si ça marche` mais pas avec au `test ça marche`.

Mots facultatifs {}

Parfois, vous pouvez vous attendre à ce qu'un mot soit ou pourrait ne pas être indiqué. Votre Modèle peut anticiper cela, faire comme si il était présent. `{``}` est proche du choix `[``]`, sauf que sa correspondance est facultative. Il est autorisé à ne pas trouver le mot.

```
?: ( C'est chaud une température de ~nombre {degré deg} Farenheit ) Ça a l'air chaud.

S: ( définissez {le mot (la signification de)} *1 > ) Désolé. Je ne sais pas.
```

Notez comment nous n'avons pas eu à dire des degrés au pluriel dans la liste facultative, car cela est automatiquement compris en utilisant le degré canonique.

Si vous souhaitez tester une séquence de mots facultatifs, vous pouvez faire deux choses:

```
u: ({rentrez} {vous} {vers} la maison)
```

ou

```
u: ({"aller vous à"} la maison)
```

Dans une phrase entre guillemets, vous pouvez utiliser les formes canoniques ou les formes exactes. Et vous pouvez faire une phrase entre guillement de 4 mots (mais pas plus).

Les commandes

Vous pouvez passer des commandes au système (le préfixe est `:`) pour vous renseigner sur des choses, contrôler des choses, déboguer des choses, etc. Dans cette section simple, nous allons examiner les commandes servant à examiner les mots et leurs relations avec eux-mêmes et les concepts. Toutes les commandes sont invisibles pour le chat normal en ce sens qu'elles n'affectent pas l'état de traitement de l'utilisateur. Une liste de toutes les commandes peut être obtenue en tapant `:commands`.

La documentation sur la plupart d'entre elles se trouve dans le manuel ChatScript Debugging Manual.

```
:word mot
```

Affiche le dictionnaire et les informations sur les faits et les concepts liés au mot. Il affiche tout ce que le système connaît sur le mot donné - ses expressions, les attributs comme s'il s'agit d'un nom singulier, quelles sont ses significations dans le dictionnaire, et à quels Eléments et Faits il est directement lié. Tapez simplement quelque chose comme `:word tennis`

`:up mot`

Alors que `:word` est intéressant, dans le but de voir ce qui correspond, la commande `:up` est plus utile, car elle vous indique comment ce mot participe dans les ensembles et dans la hiérarchie héritée à la fois des concepts et du Net, ainsi, l'ensemble des relations répertoriées serait reconnu si le mot donné comme argument est utilisé.

Supposons que vous créez le concept de `~bâtiments`. Il suffit de penser à un mot que vous voulez y inclure, comme *Temple*,

```
>>:up temple

For temple:
 Set hierarchy:
    ~religious_buildings
       ~building
          ~tool
          ~locatedentity
             ~there
          ~immovable
             ~functions
          ~enterable
          ~artifacts
             ~objects
                ~nounlist
             ~human_data
 Wordnet hierarchy:
  temple~1:N    the place of worship for a Jewish congregation
    is house_of_worship~1  any building where congregations gather for prayer
       is building~3 a structure that has a roof and walls and stands more or
less permanently in one place
          is structure~1 a thing constructed
             is artifact~1 a man-made object taken as a whole
                is whole~1 an assemblage of parts that is regarded as a single
entity
                ~entirety_words
                ~goodness
                   is object~1 a tangible and visible entity
                   ~verbal_disagree
                      is physical_entity~1 an entity that has physical existence
                         is entity~1 that which is perceived or known or
inferred to have its own distinct existence
                         ~nounroot
  temple~2:N    place of worship consisting of an edifice for the worship of a
deity
    is house_of_worship~1  any building where congregations gather for prayer
  temple~3:N    an edifice devoted to special or exalted purposes
    is building~3  a structure that has a roof and walls and stands more or less
permanently in one place
  temple~4:N    the flat area on either side of the forehead
    is feature~2  the characteristic parts of a person's face: eyes and nose and
mouth and chin
```

```
~focus
   is body_part~1 any part of an organism such as an organ or extremity
      is piece~11 a portion of a natural object
      ~unitmeasure
         is thing~1 a separate and self-contained entity
            is physical_entity~1 an entity that has physical existence
```

Si nous essayons de construire le concept de bâtiments, alors le `temple~1`, le `temple~2` et le `temple~3` sont des définitions qui ont du sens (:N ne nomme qu'une expression). Mais notez qu'au début de l'affichage que ces définitions proviennent toutes de `building~3`, et que l'utiliser aurait du sens et engloberait tout ce que le WoldNet considérerait comme un bâtiment dans ce sens.

`:down mot limit`

prend un mot et poursuit sa hiérarchie en affichant ce qui hérite de lui. La limite combien de niveaux jusqu'où aller (la valeur par défaut est 2) car la digression peut se développer très loin pour certains mots. Si le mot est un concept ou un nom de Topic, il affiche ses membres de haut niveau. `:down entité 1` ou `:down ~animaux 2`.

SORTIE SIMPLE

L'objectif pour le moteur est de générer des résultats à afficher à un utilisateur. Quand une Règle le fait, elle a atteint l'objectif du Topic.

Sortie directe

Pour générer une sortie simple, mettez simplement le texte que vous voulez afficher après le Modèle d'une Règle. Les Gambits ne doivent pas comporter de composant d'un Modèle, auquel cas leur affichage a lieu immédiatement.

```
t: Ceci est délivré pour l'utilisateur.
```

```
?: ( salut ) comment allez-vous? Vivez-vous ? Allez-vous mourir bientôt ?
```

Auto formatage

Vous donnez les mots et la ponctuation pour l'affichage. Le système les formate automatiquement, peu importe si les virgules, les espaces et les tabulation qui les précèdent, ou combien de mots-clés ou d'onglets existent entre les mots. Le système les reformate automatiquement. `Je vous aime bien?` et `Je vous aime bien ?` Seront affichés de la même façon.

Si vous avez réellement besoin de contrôler l'espacement, consultez les « formatages avec guillemets» (formatted double quotes) dans le manuel

utilisateur avancé ChatScript Advanced User Manual.

Sortie littérale \

Pour obtenir une sortie de caractères réservé ayant un sens pour le moteur, comme **[** et **]** , vous devez mettre une barre oblique inverse juste devant eux : `\`. A l'inverse, pour afficher une nouvelle ligne, vous utiliserez `\n` et une tabulation avec `\t`.

Sortie aléatoire []

Vous pouvez sélectionner parmi une gamme de choix en utilisant des choix de sortie. Chaque choix est placé entre [], et un ensemble contigu de ceux-ci forment une zone dans laquelle le système choisira au hasard. Chaque fois que les articles sont présentés de façon discontinue, vous obtenez une nouvelle zone aléatoire.

```
?: (salut) [bonjour.] [salut] [hey] Tu va [danser] [nager] [manger] prochainement ?
```

Dans ce qui précède il y a deux zones aléatoires, séparées par un texte fixe. Ainsi, il pourrait produire

```
hey Tu vas danser prochainement ?
```

Ou

```
bonjour. Tu va manger prochainement ?
```

Les variables

Un chatBot avec aucune capacité de se souvenir, même brièvement, de ce que disent les utilisateurs, serait pauvre. ChatScript prend en charge plusieurs niveaux de mémorisation. La variable ultime crée des Faits, mais elle fait l'objet de son propre manuel ChatScript Fact Manual.

_* la variable correspondante

Lorsque vous utilisez des Joker et définissez un Modèle, vous pouvez demander au système de mémoriser brièvement le mot qui correspondra. Il suffit de placer un trait de soulignement devant ce que vous voulez mémoriser.

Le but de la mémorisation est de pouvoir utiliser sa valeur sur une sortie. Les résultats de la mémorisation sont stockés comme Variables Correspondante nommées `_0`, `_1`, etc., selon le nombre de soulignement que vous aurez utiliser

dans votre Modèle.

```
?: (tu manges *~2 _~viande) Non, je déteste _0.
```

Si l'entrée est-ce que `tu manges du jambon`, la sortie serait `Non, je déteste le jambon`. Bien sûr, la valeur de `_0` n'est garantie que pour l'exécution de cette Règle. Les variables de correspondance peuvent être effacées lorsque vous exécutez une autre règle. Ou ils peuvent durer un moment.

Au plus, il durera la durée de cette volée (plusieurs phrases peut-être) après quoi il devrait être présumé détruit. Chaque fois que vous commencez une volée, vous devriez présumer que les variables de correspondance contiennent des résidus inconnus.

ChatScript enregistre ces variables de `_0` à `_20`.

J'utilise souvent la gamme de `_10` à `_20` comme variables « de sécurité » pour la durée d'une volée, car je n'ajouterai jamais autant de variables dans une seule phrase. Il est peu probable que je dépasse 5. Je peux donc les utiliser comme je le souhaite.

Lorsque le système mémorise la variables correspondant à votre _ , il stocke à la fois le mot original, sa forme canonique et sa position dans le texte. En sortie, par défaut, vous obtenez la forme canonique. Si vous voulez utiliser la forme originale, vous devez faire précéder votre référence d’une apostrophe.

```
?: (vous mangez _ [jambon oeufs bacon]) Je mange '_0.
```

Si l'entrée est : `vous mangez des œufs`, la sortie sera `je mange des œufs`. Si je n'avais pas utilisé l'apostrophe, la sortie aurait été : `je mange un œuf`.

Exceptionnellement, vous pourriez avoir besoin d’utiliser plusieurs formes canonique mémorisée.

```
?: (Aimez-vous _* ou _*) Je n'aime pas '_0 donc je suppose que cela signifie que je préfère '_1.
```

Si l'entrée est que vous aimez manger des œufs verts ou nager sur la plage, le résultat serait que je n'aime pas manger des œufs verts, donc je suppose que cela signifie que je préfère nager sur la plage.

Si vous mémorisez une zone optionnelle, {comme ça}, vous obtiendrez soit le mot correspondant, soit la variable match sera définie sur Null si elle ne correspond pas. Une variable nulle n'imprime rien en sortie.

Si vous utilisez des variables correspondantes, elles sont allouées dans l'ordre du modèle. Par exemple.,

```
s: (_~fruit [_~animal _ours] _~aime)
```

Dans ce qui précède, _0 est un fruit et _2 est un « aimer », et le _~animal ou le _ours est _1.

Si vous n'aviez PAS mis _ devant l'ours, vous preniez le risque que ~aime soit enregistré comme _1 ou _2, selon ce qui s'est passé à l'intérieur des []. C'est le mal de crâne assuré si vous utilisez des mémorisations imbriquées.

Soyez conscient que :

u: (_1)

ne mémorise pas le mot d'entrée "1". Il nomme plutôt la variable correspondante 1 et demande si elle a une valeur. Si vous vouliez à tout prix mémoriser le chiffre 1 (chose étrange), vous auriez à faire comme ceci:

u: (_~nombre _0 = 1)

$ Variables_Utilisateur

Si vous avez besoin d'une mémoire qui dure au-delà de l'entrée actuelle, une possibilité est d’utiliser les variables utilisateur. Le nom dune variable commence par un signe de dollar ou deux, puis suit une lettre alphabétique, ensuite le reste doit être alphabétique, chiffre, trait de soulignement ou trait d'union. Vous l'initialisez en utilisant une affectation de style C dans la sortie.

L'opérateur d'assignation = DOIT être séparé de la variable et de la valeur par au moins un espace, sinon le système n'a aucun moyen de dire que vous ne voulez pas qu'il produise simplement un mot bizarre.

Contrairement aux Variables correspondantes, les Variables utilisateur ne contiennent qu'une seule valeur.

s: (Je mange _*1>) $food = '_0 Je mange des huîtres.

Nous vous conseillons de placer ces scripts de calcul sur des lignes séparées afin de faciliter la lecture de votre script, mais ChatScript ne s'en soucie pas vraiment.

s: (Je mange _*1>)

$ food = '_0

Je mange des huîtres.

Les variables dureront toujours ou jusqu'à ce que vous les changiez. Si vous voulez que la variable disparaisse à la fin de la volée, nommez-la avec $$ au début, par exemple. $$nourriture. En plus de l'affectation simple, vous pouvez faire +=, -=, *=, /=, %=, |=, &=, ^=, et |^=, par exemple

$$MaVar += 100.

|^= désactive les bits et $x |^= 2 équivaut à $x &= (-1 ^ 2)

Les affectations de variables s'étendent également entre les opérations arithmétiques, mais vous ne pouvez pas utiliser les parenthèses pour contrôler la priorité des opérateurs. Par exemple :

$MaValeur = $foo + 20 * 5 / 59 Il s'agit d'une sortie normale après l'affectation.

Vous pouvez même assigner des ensembles de faits de diverses manières (voir le ChatScript Fact Manual pour comprendre les faits) comme:

@2 = @3 # mettre tous les faits en 3 en 2

@2 += @3 # augment 2 avec des faits de 3 (permet des doublons)

@2 -= @3 # supprime les faits de 3 à partir de 2

@2 = $$factid # prends un ID de fait et le mets comme le seul contenu de 2

@2 += $$factid # ajoute l'id du fait à 2

@2 -= $$factid # retranche 2 à l'id du fait

ATTENTION:

Faites attention à l'arithmétique étendue. Chaque opération s'applique au résultat de la dernière.

$myvalue += 2*4 signifie ($myvalue + 2) * 4.

$tmp =% heure + $ tmp signifie (2 *% heure)

Bien sûr, il aurait été plus clair d'écrire ceci:

$myvalue = $foo + 20 * 5/59

Il s'agit d'une sortie normale après l'affectation. Vous ne pouvez pas utiliser le nom de la variable que vous affectez dans la partie droite, car l'arithmétique

est effectuée progressivement sur les termes, de sorte que la variable est remplacée au premier terme. De même, vous pourriez être surpris par quelque chose comme ceci:

$myvar -= $articlesize * 10

car il déduit d'abord $articlesize de $myvar, puis multiplie ce résultat par 10.

Vous pouvez tester les variables dans les modèles de plusieurs façons. Certains de ces tests n'affectent pas le pointeur de position actuel de la correspondance. En mettant simplement le nom de la variable on obtiendra : "a-t-elle une valeur"?

?: (quel est mon nom $nom) Votre nom est $nom.

Comme les variables correspondantes, vous pouvez utiliser des variables utilisateur dans la sortie. Ici, si l'entrée est quel est mon nom et si $nom a déjà été assigné, alors le Modèle correspond. Sinon, il échoue. Vous êtes libre de vous référer à des variables qui n'existent pas. Le modèle échouera, tout simplement.

Vous pouvez également tester directement une variable pour voir si elle a une valeur particulière en utilisant un test relationnel dans le Modèle. Les tests relationnels n'utilisent aucun espace, ils sont tous un seul gros Token. Une telle relation est =, aussi inscriptible en tant que ==.

?: ($genre =homme J'aime les garçons) Oh, mon cher !

Ici, la règle commencera seulement à vérifier les correspondances d'entrée si $genre a la valeur Homme (insensible à la casse). Si elle n'est pas définie ou a une autre valeur, cette règle échoue immédiatement. Un test relationnel exige que les deux parties de la relation et le symbole de relation soient tous collés sans espace. Donc, la règle suivante équivaut à voir si ‘$genre a déjà été assigné, puis voir si l'utilisateur a tapé =homme’ à n’importe quel moment.

?: ($genre =homme)

Les autres relations sont < et >, ce qui nécessite que le système convertisse la valeur de texte de la variable en numérique.

?: (J’ai _~nombre ans _0<10) Tu es un gamin

Vous pouvez inverser un test de relation en utilisant l’opérateur « ! ».

?: (J’ai _~nombre ans !_0<10) Tu es un gamin

Pour les tests d'égalité, vous pouvez utiliser l'opérateur « != ».

?: ($var!=5) OK.

Vous pouvez assigner manuellement des variables de correspondance, mais à moins que vous ne les affectiez à partir d'une variable correspondante existante, vous n'obtiendrez aucune donnée canonique ou position de la correspondance.

s: (ma vie) _8 = bonjour

s: (mon _*1) _8 = _0

Dans le premier exemple, _8 reçoit le mot bonjour dans s forme canonique et originale. Il ne le traite pas. Dans le second exemple, puisque _0 a une forme double avec une position, l'affectation est double et passe sa position.

Effacer les variables

Vous pouvez effacer une variable utilisateur ou une variable correspondant en la définissant sur null:

$myvar = null

_3 = null

Variables pour le long terme

Le système stocke normalement les variables par utilisateur. Vous pouvez définir des Faits spécifiques au bot au moment où vous vous connectez à lui. Si vous avez des Faits que vous voulez passer globalement à l’ensemble des Bots et dans le cadre du système de base, vous pouvez placer ces assignations dans une table et les lire sous une commande :build.

Allez lire le manuel d'utilisation de ChatScript pour en savoir plus sur les faits.

% Les Variables système

Le système possède des variables prédéfinies que vous pouvez généralement

tester et utiliser, mais que vous ne pouvez normalement pas affecter. Elles commencent par %. Ceux-ci incluent %hour, %bot et autres.

Voir Variables système ChatScript et Concepts définis par le moteur.

Résumé

C’est pas aussi simple que tu l'aurais voulu. Sans doute, je vous en ai dit beaucoup plus que ce que vous vouliez vraiment en savoir à ce stade, mais cela met la scène en place pour que vous sachiez qu'il y a beaucoup de possibilités (bien que nous les ayons à peine effleurées ici). Rappelez-vous, pour commencer, tout ce dont vous avez besoin est d'écrire un Topic, avec des mots clés, des Gambits triviaux, des Répondeurs et des Répliques avec des motifs simples, et une sortie qui est exactement ce que vous voulez que le bot dise.

[Wiki home] - [Manuel de l'utilisateur avancé]