Module d'Intelligence Artificielle
Mini-projet sur trois séances de TP
Implémentation du chatbot ELIZA en LISP

Christophe Denis

Mercredi 22 janvier 2025

Conditions d'utilisation du document

Droits accordés par cette licence :

- Partager le document en mentionnant l'auteur : Christophe Denis (christophe.denis@sorbonne-universite.fr).
- Utiliser l'œuvre à des fins personnelles ou non commerciales, sans modification.

Conséquences du non-respect de la licence :

- Violation des droits d'auteur, pouvant entraîner des poursuites judiciaires et des sanctions légales, conformément à l'article L335-2 du Code de la Propriété Intellectuelle.
- Suppression de l'accès au document.

Présentation du mini-projet

Objectifs du mini-projet

- Comprendre et implémenter un chatbot inspiré d'ELIZA.
- Développer des compétences en correspondance de motifs en Lisp.
- Apprendre à structurer et documenter un projet en programmation.

Organisation :

- **Travail en binôme recommandé** pour favoriser la collaboration.
- Un **rapport de 5 pages** est attendu à la fin du projet.

Déroulement du mini-projet (6 heures)

Organisation en trois séances :

- **Séance 1 (2h)** : Découverte et mise en place.
- **Séance 2 (2h)** : Implémentation des fonctionnalités.
- **Séance 3 (2h)** : Tests et rédaction du rapport.

Rendus :

- Code fonctionnel du chatbot ELIZA.
- Rapport détaillant les étapes de conception et de test.

Organisation des séances

A titre indicatif. Une quatrième séance pourrait être programmée.

- **Séance 1 :** `match-eq`, `match`, `wildcard`, `random-elt`
- **Séance 2 :** `bind`, `subs`, `swap`
- **Séance 3 :** Intégration avec `eliza`

Architecture et principe de fonctionnement

Intrication des fonctions (1/3)

Objectif : Comprendre l'enchaînement des fonctions pour générer une réponse.

Étape 1 : Entrée utilisateur

- L'utilisateur entre une phrase : { `"I feel sad"` }
- La fonction `eliza` transforme cette entrée en liste de symboles : `(I FEEL SAD)`

Étape 2 : Correspondance de motifs

- La fonction `match` compare l'entrée utilisateur avec les motifs définis dans `*rules*`.
- La fonction `wildcard` identifie les parties variables de l'entrée utilisateur et les stocke.
- Exemple de correspondance trouvée : `((* x I feel * y) (do you often feel y ?))`

Étape 3 : Création des liaisons

- La fonction `bind` enregistre les parties capturées dans `*bindings*`.
- Exemples de liaisons stockées : `((X NIL) (Y SAD))`

Intrication des fonctions (2/3)

Étape 4 : Substitution des variables

- La fonction `lookup` est utilisée pour rechercher les valeurs dans `*bindings*`.
- La fonction `subs` remplace les variables dans le modèle de réponse. Exemple : `(DO YOU OFTEN FEEL Y ?)` devient `(DO YOU OFTEN FEEL SAD ?)`

Étape 5 : Changement de perspective

- La fonction `swap` ajuste la réponse pour adopter la perspective correcte.
- Exemple d'ajustement : `(I WANT MY COMPUTER)` $\Rightarrow$ `(YOU WANT YOUR COMPUTER)`
- Cette transformation est réalisée en consultant la table de substitution `*viewpoint*`.

Intrication des fonctions (3/3)

Étape 6 : Sélection d'une réponse

- La fonction `random-elt` sélectionne une réponse aléatoire parmi celles disponibles.
- Exemple de réponses possibles : `(DO YOU OFTEN FEEL Y ?)` `(WHY DO YOU THINK Y ?)`

Étape 7 : Affichage de la réponse

- La fonction `eliza` affiche la réponse finale à l'utilisateur.
- Exemple de réponse affichée : `"Do you often feel sad ?"`

Résumé du processus :

- Entrée utilisateur → `eliza`
- Correspondance → `match` + `wildcard`
- Stockage → `bind` + `lookup`
- Substitution → `subs` + `swap`
- Sélection → `random-elt`
- Réponse affichée → `eliza`

Fonctions LISP à développer

Fonction `eliza` (Séance 3)

Prototype :

```
(defun eliza ())
```

Objectif :

- Lire l'entrée utilisateur en boucle.
- Appliquer les règles définies et générer une réponse.
- Quitter lorsque l'utilisateur tape "bye".

Exemple d'utilisation :

```
> (eliza)
} hello
hello. how can i help ?

} bye
```

Fonction `match-eq` (Séance 1)

Prototype :

```
(defun match-eq (pat in) -> t/nil)
```

Objectif :

- Vérifier si deux listes sont strictement identiques sans caractères génériques.

Exemple d'utilisation :

```
> (match-eq '(i feel happy) '(i feel happy))
t

> (match-eq '(i feel sad) '(i feel happy))
nil
```

Fonction `match` (Séance 1)

Prototype :

```
(defun match (pat in) -> t/nil)
```

Objectif :

- Comparer un motif à une entrée utilisateur avec gestion des jokers (*).

Exemple d'utilisation :

```
> (match '(i feel *) '(i feel happy))
t
> *bindings*
((X HAPPY))
```

Fonction `wildcard` (Séance 1)

Prototype :

```
(defun wildcard (pat in) -> t/nil)
```

Objectif :

- Gérer les caractères génériques * et assigner les segments aux variables.

Exemple d'utilisation :

```
> (wildcard '(* x i hate * y) '(sometimes i hate
    my job))
t
> *bindings*
((X SOMETIMES) (Y MY JOB))
```

Fonction bind (Séance 2)

Prototype :

```
(defun bind (var value bindings) -> list)
```

Objectif :

- Ajouter une valeur associée à une variable dans la liste de liaisons *bindings*.

Exemple d'utilisation :

```
> (bind 'x 'cat '((x dog) (y sheep)))
((X CAT DOG) (Y SHEEP))
```

Fonction `lookup` (1/2) (Séance 2)

Prototype :

```
(defun lookup (key alist) -> value)
```

Objectif :

- Rechercher une valeur associée à une clé dans une liste d'associations (`alist`).
- Utilisée dans les fonctions `subs` et `swap` pour récupérer des valeurs stockées.

Explication :

- La fonction prend en entrée :
  - Une **clé** à rechercher.
  - Une **liste d'associations** de la forme : `'((X DOGS) (Y CATS AND SHEEP))`.
- Elle parcourt récursivement la liste et retourne la valeur associée à la clé.
- Si la clé n'est pas trouvée, la fonction retourne `nil`.

Fonction lookup (2/2)

Exemples d'utilisation :

```
> (setq *bindings* '((x dogs) (y cats and sheep)
   ))

> (lookup 'x *bindings*)
(X DOGS)

> (lookup 'y *bindings*)
(Y CATS AND SHEEP)

> (lookup 'z *bindings*)
NIL
```

Fonction subs (Séance 2)

Prototype :

```
(defun subs (list) -> list)
```

Objectif :

- Remplacer les variables dans une liste en utilisant les liaisons définies dans *bindings*.

Exemple d'utilisation :

```
> (setq *bindings* '((X DOGS) (Y CATS AND SHEEP)
   ))
> (subs '(I think y are chased by x))
(I THINK CATS AND SHEEP ARE CHASED BY DOGS)
```

Fonction swap (1/2) (Séance 2)

Prototype :

```
(defun swap (value) -> value)
```

Objectif :

- Remplacer certains mots pour changer de perspective dans la conversation.
- Utiliser la table de substitution `*viewpoint*`.

Explication :

La variable globale `*viewpoint*` contient des paires de mots pour changer de perspective, par exemple :

```
(defvar *viewpoint* '((I YOU) (YOU I) (MY YOUR)
    (YOUR MY)
                          (AM ARE) (WAS WERE)))
```

Ainsi, lorsqu'on échange `'I'` dans une phrase, il sera remplacé par `'YOU'`, et inversement.

Fonction swap (2/2) (Séance 2)

Exemple d'utilisation :

```
> (swap 'i)
YOU

> (swap 'my)
YOUR

> (swap 'was)
WERE

> (swap 'computer)
COMPUTER ; Aucun changement si le mot n'est pas
    dans *viewpoint*
```

Fonction random-elt (Séance 1)

Prototype :

```
(defun random-elt (list) -> element)
```

Objectif :

- Sélectionner aléatoirement un élément d'une liste de réponses.

Exemple d'utilisation :

```
> (random-elt '(yes no maybe))
MAYBE

> (random-elt '("Tell me more." "Why do you
    think so?"))
"WHY DO YOU THINK SO."
```

Variables utilisées

Variable *bindings*

Rôle :

- Stocke les liaisons entre les variables capturées par les motifs et leurs valeurs extraites de l'entrée utilisateur.

Initialisation possible :

```
(setq *bindings* '((X DOGS) (Y CATS AND SHEEP)))
```

Fonctions qui l'utilisent :

- `match` – Capture les variables et les stocke.
- `wildcard` – Met à jour les liaisons.
- `bind` – Ajoute ou met à jour des valeurs dans `*bindings*`.
- `subs` – Remplace les variables dans les réponses.
- `eliza` – Réinitialise `*bindings*` à chaque nouvelle entrée.

Variable *viewpoint*

Rôle :

- Contient les paires de mots utilisés pour changer de perspective dans les réponses du chatbot.

Initialisation possible :

```
(defvar *viewpoint* '((I YOU) (YOU I) (MY YOUR)
                      (YOUR MY) (AM ARE) (WAS
                         WERE)))
```

Fonctions qui l'utilisent :

- `swap` – Effectue la substitution en fonction des paires de `*viewpoint*`.
- `bind` – Applique la substitution lors de l'ajout de valeurs.

Variable *rules*

Rôle :

- Contient l'ensemble des motifs et des réponses possibles pour Eliza.

Initialisation possible :

```
(defparameter *rules*
  '(((* x hello * y) (hello. how can I help ?))
    ((* x i want * y) (what would it mean if you
        got y ?)
                      (why do you want y ?))
    ((* x i hate * y) (what makes you hate y ?))
       ))
```

Fonctions qui l'utilisent :

- `eliza` – Parcourt les règles pour trouver une correspondance.
- `match` – Compare l'entrée utilisateur avec les motifs de `*rules*`.
- `random-elt` – Sélectionne aléatoirement une réponse dans les règles associées.

Variable *input*

Rôle :

- Stocke l'entrée utilisateur sous forme de liste de symboles après transformation.

Initialisation possible :

```
(setq *input* '(I FEEL SAD))
```

Fonctions qui l'utilisent :

- `eliza` – Initialise et analyse l'entrée utilisateur.
- `match` – Compare l'entrée avec les motifs.

LISP : Langage Infiniment Sublime Parenthèsiques

Pourquoi coder en LISP ?

- Parce que la vie, c'est aussi une série de parenthèses bien fermées `()`.
- Parce que taper `(+ 1 1)` est plus philosophique que `1 + 1`.
- Parce qu'un bon programmeur LISP est celui qui ne se perd pas dans une forêt de parenthèses.

Les 3 phases d'apprentissage de LISP :

1. Déni : "Pourquoi autant de parenthèses ?"
2. Colère : "Trop de parenthèses !"
3. Acceptation : "Les parenthèses, c'est la vie."

"LISP est à la programmation ce que le zen est à la méditation... il faut d'abord accepter le vide (des parenthèses)."

Un programmeur LISP heureux ?

(HAPPY (CODING LISP))