Intelligence Artificielle : **IA01**

Automne 2015

Table des matières

Chapitre 1

La notion d'Intelligence Artificielle

1.1 Définition et histoire de l'IA

Cette section ne semble pas très utile, mais elle peut tomber à un médian.

1.1.1 Définition

La notion d'*Intelligence Artificielle* est composée de deux mots : *Intelligence* et, bien sûr, *Artificielle*.

L'*Intelligence* signifie pouvoir *saisir des connaissances*, les *appliquer/adapter* à différentes situations mais aussi *conceptualiser* le monde qui nous entoure.

Lorsque on associe l'*Intelligence* à *Artificielle*, on obtient non pas un ordinateur pensant mais une activité *d'acquisition de connaissances*. Par le biais de l'IA, les ordinateurs obtiennent donc des connaissances qu'ils pourront utiliser *intelligemment*.

1.1.2 Histoire de l'IA

à faire

1.2 Les approches à l'Intelligence Artificielle

Aujourd'hui, il existe principalement 4 approches à la création d'une IA : l'approche **symbolique**, l'approche **numérique**, l'approche **hybride** et l'approche **distribuée**.

1.2.1 Approche symbolique

L'approche symbolique vise à étudier le raisonnement, et se base donc sur des raisonnements créés, pensés par les programmeurs de l'IA.
En effet, dans cette approche, l'IA suit des processus de raisonnement qui ont étés formalisés - et codés - par le programmeur. Par conséquent, dans ce type d'approche, il faut représenter de façon explicite le problème et ces méthodes de résolution.
Un exemple utilisant cette approche peuvent être les *systèmes experts*.

1.2.2 Approche numérique

Dans cette approche, on étudie la perception et les réflexes face à un problème. Tandis que l'approche symbolique utilise des raisonnements créés explicitement, l'approche numérique exploite la notion d'*apprentissage par l'exemple*. En effet, il y a peu de programmation explicite, et ces IA se basent sur des modèles informatiques. On a donc des systèmes qui évoluent et s'adaptent automatiquement.

1.2.3 Approche hybride

Cette approche utilise les deux approches décrites ci-dessus.

1.2.4 Approche distribuée

L'approche distribuée se différencie des autres par rapport à la relation qu'entretient un système avec les autres. Effectivement, l'IA traditionnelle fonctionne de manière indépendante, sans contexte ni communication avec d'autres systèmes. Le but de l'approche distribuée est donc de créer des systèmes composés de multiples sous-systèmes, qui interagissent entre eux.

Chapitre 2

Programmation Lisp

Lisp, aka *List Processor* est le langage principalement utilisé pour réaliser les différents TD/TP que nous tend l'UTC. Bien qu'il soit ancien, il n'est pas à mettre entre parenthèses [1].
Ainsi, il sera question dans ce chapitre des différentes fonctionnalités vues à propos de ce langage [2], plutôt que de la représentation même de connaissances.

2.1 Principes de programmation

Lisp diffère pas mal des autres langages sur sa manière d'être programmé.
Par rapport à un langage bien connu comme le C, Lisp se programme de façon *interactive*, *fonctionnelle* et *symbolique*.

2.1.1 Interactivité

L'*interactivité* signifie qu'il n'y a pas de compilation préalable avant l'exécution : un fichier code Lisp est donc un ensemble de commandes qui seront directement interprétés par un *interpréteur Lisp*.

2.1.2 Fonctionnel

La programmation *fonctionnelle* est un paradigme de programmation, au même titre que la programmation orienté objet, par exemple.
Comme son nom l'indique, la programmation fonctionnelle se base surtout sur l'utilisation de fonctions qui s'appellent entre elles. Ainsi, un programme Lisp n'aura pas de structure fixe, contrairement à un programme C qui commencera par exécuter *main()*. Un programme Lisp sera donc principalement composé de fonctions qui peuvent être récursives.

2.1.3 Symbolisme

Le *symbolique* signifie quant à lui que l'on manipule pas seulement des *chiffres* ou *caractères*, mais une sorte d'information plus générale que ça : les symboles.
Il n'y a donc pas de *type* à proprement parler : pas de *int, float, double, char, ...* mais que des symboles.

1. ha ha ha ha ha
2. plus précisément le Common Lisp

2.2 Concepts basiques du Lisp

2.2.1 Objet de base : l'atome

Un atome est soit un nombre, soit un symbole.
La différence entre un nombre et un symbole tient dans le fait qu'un symbole est une séquence non vide de caractères qui *pointe* vers un objet - un objet pouvant être une donnée, ou encore une fonction.

2.2.2 Structure de base : la liste

La liste est la principale brique du *Lisp*. Une liste entre parenthèses, et sert principalement à lancer des fonctions, ou sinon à... lister des items.
Par exemple :

```
(liste avec des objets)
```

Dans cette liste, on a donc les objets

```
liste avec des objets
```

Mais les listes servent surtout à appeler des fonctions. Par exemple, si on veut appeler la fonction *merci* avec les arguments *oui,1*, il faudra l'appeler de cette manière :

```
(merci oui 1)
```

2.2.3 Valeurs logiques / Prédicats

Il existe en Lisp des fonctions qui retournent vrai ou faux ; ce sont les *prédicats*. Ces prédicats servent principalement à vérifier le type d'une variable. Les prédicats principalement utilisés sont ici *numberp, listp, floatp, integerp, null, symbolp, stringp*.
Il existe aussi des *semi-prédicats*, qui sont des fonctions qui renvoient soit faux, soit une valeur considérée vraie [3].
member, *and/or*, sont des semis-prédicats.

2.2.4 Définition de fonction

Il est possible de définir une fonction de deux manières :
— avec *defun*, qui permet d'avoir une fonction nommée ;
— avec *lambda*, qui permet d'avoir une fonction anonyme ;

2.3 Variables, portée, visibilité

Les variables peuvent avoir deux portées différentes : *lexicale* ou *dynamique*.

2.3.1 Variables lexicales

Une **variable lexicale** est une variable qui a une portée *locale*, dans le sens où elle ne sera vue que dans un contexte donné : là où elle aura été définie.
Un exemple implicite de variables lexicales sont les paramètres d'une fonction. Prenons par exemple la fonction suivante :

3. Tout ce qui n'est pas NIL est vrai en Lisp

```
(defun test(x)
  (print x)
)
```

Il parait évident que le parametre x n'existera que dans le contexte de la fonction *test*, et non pas en dehors de cette fonction.
Il est à noter que les variables lexicales ne sont vues qu'à un niveau. Par exemple, si on avait une autre fonction *test*2 dans notre fonction *test* ci-dessus, et qu'elle essaye d'accéder à la variable x de la fonction *test*, ça ne fonctionnerait pas.

Variables locales

Les variables *locales* en Lisp sont les variables définies par les fonctions *let/let**. En gros, ce sont des variables qui sont uilisées qu'au sein d'une fonction.
Par défaut, les variables locales sont des variables **lexicales**.

Variables dynamiques Les variables dynamiques sont des variables à la visibilité globale.
C'est à dire qu'une variable dynamique sera vue de toutes les sous fonctions d'une fonction, etc ...

Variables globales

Une variable globale est donc une variable dynamique.
Pour définir une variable globale, il faut utiliser l'opérateur *defvar*, ou *defparameter*, avec la différence que *defparameter* réinitialise la variable avec la valeur donnée même si elle existe déjà.

2.4 Structures de contrôles / Blocs

2.4.1 Opérateur conditionnel : *if*

L'opérateur *if* se présente de la manière suivante :

```
(if [condition]
  [truc a executer si condition vraie]
  [truc a executer si condition fausse]
)
```

L'opérateur *if* ne permet donc que d'exécuter qu'une seule commande.

2.4.2 Blocs conditionnels : cond/when/unless

Ces 3 blocs répondent à la limite principale du *if* qui ne permet de n'exécuter qu'une seule commande.
Ces 3 structures permettent en effet d'exécuter des blocs d'instruction, plûtot qu'une seule instruction.
La différence entre ces 3 blocs tient dans les conditions : *cond* permet de faire plusieurs tests, *when* n'en teste qu'une seule et *unless* correspond à un *sauf si*.
Syntaxiquement, ces blocs s'utilisent de la manière suivante :

```
; cond
(cond
  ([test1]
    ([action1]
     [action2]
    )
  )
```

```
  ([test2]
   ([action1]
    [action2]
   )
  )
)

; when
(when [test]
  [action1]
  [action2]
)

; unless
(unless [test]
  [action1]
  [action2]
)
```

2.4.3 Blocs itératifs