

Brève histoire de l'Intelligence Artificielle

Olivier Boisard,

Ingénieur-conseil et enseignant à l'École Centrale de Lille

olivier@olivier-boisard.net , www.olivier-boisard.net

Mars 2020

Résumé :

L'invention de la machine de Turing, en 1936, rend possible le développement de l'informatique et de l'Intelligence Artificielle (IA). Les premiers ordinateurs seront toutefois plutôt destinés au calcul numérique ou à la gestion de bases de données. Il faudra attendre les années 1970 pour que l'IA prenne véritablement son essor, avec l'apparition de nouveaux langages de programmation et de logiciels tels que les systèmes experts. Aujourd'hui, ce sont les réseaux neuronaux qui s'imposent, et cette réussite soulève des questions sur l'usage de l'IA, qui dépassent très largement le seul cadre de l'informatique.

Des machines destinées à simuler les comportements humains ont été imaginées bien avant la création de la robotique et de l'informatique. Mais il s'agissait alors d'automates, dont chaque action, ou chaque réaction dans un contexte donné, devait être préalablement pensée et intégrée explicitement dans des mécanismes complexes. Le XVIII^e siècle, et plus encore le XIX^e - parfois désigné comme *l'âge d'or des automates* - ont vu se multiplier ces machines étranges et spectaculaires.

Un exemple emblématique de cet âge d'or est le célèbre automate joueur d'échec - le « *Turc Mécanique* » - présenté au public durant des décennies en Europe puis en Amérique, et dont on raconte qu'il provoqua en 1809 la colère de Napoléon - pourtant réputé bon joueur d'échec - furieux d'avoir dû concéder une défaite cuisante.

Mais, comme le démontre Edgar Poe dans la nouvelle *Le joueur d'échec de Maezel*¹, cet automate ne pouvait être qu'une imposture, et derrière des mécanismes d'apparence se cachait nécessairement un joueur humain ...

Pourtant, dans sa volonté de démontrer rationnellement l'impossibilité d'une telle machine, Edgar Poe ne voit pas assez loin. Aujourd'hui, le moindre ordinateur a la capacité de faire tourner un logiciel dont les performances, pour limitées qu'elles soient aux yeux d'un grand joueur d'échec, sont suffisantes pour mettre en difficulté nombre d'amateurs.

Alors, où se situe la différence entre un automate et une machine jouant « intelligemment » ? Est-ce une simple question de puissance de calcul, désormais permise par la miniaturisation extrême des micro-processeurs ? Ou, plus fondamentalement, est-ce une différence de nature entre des mécaniques qui se bornaient à exécuter des tâches prédéfinies, et des circuits capables de faire appel à des modes de fonctionnement bien plus subtiles ?

De l'automate à la machine de Turing

Clairement, c'est bien d'une différence de nature qu'il s'agit : lorsqu'en 1936 Alan Turing conçoit sa célèbre *machine de Turing*, sont réunis tous les éléments qui donneront naissance à l'informatique contemporaine, grâce à des engins n'ayant plus rien à voir avec des automates.

Ce qui caractérise ces machines, au demeurant assez simples à décrire sur le plan théorique, est leur capacité à manipuler des symboles ayant indifféremment le statut de *données* (la *data*, autrement dit l'information qui est manipulée), ou celui de programme (le *process*, autrement dit ce qui dicte à la machine la façon de manipuler la *data*). En d'autres termes, cela revient à considérer une machine capable de modifier elle-même son propre mécanisme, soit, si cette modification est effectuée en vue d'une fin particulière, de développer *une capacité d'apprentissage*.

Un automate joueur d'échec est inconcevable car son concepteur devrait préalablement anticiper toutes les parties du jeu - leur nombre est certes fini, mais incommensurable - et traduire chacune par un mécanisme particulier. Alors qu'une machine de Turing correctement conçue a la capacité de jouer en s'adaptant à chaque nouvelle configuration de l'échiquier, sans avoir à anticiper toutes les parties possibles : comme un humain, son jeu ne sera peut-être pas parfait, mais chaque erreur se transformera en expérience acquise, permettant de mieux jouer la partie suivante. En soi, l'apprentissage et la capacité d'adaptation sont déjà deux qualités caractéristiques de l'intelligence !

L'ordinateur, machine à calculer ou machine logique ?

Parmi les précurseurs de l'informatique et de l'intelligence artificielle (IA) - terme inventé assez tardivement, en 1956, par John McCarthy et Marvin Minsky - il faut également citer des personnalités telles que Claude Shannon, créateur de la théorie de l'information, dont un des apports majeurs est le choix du *bit*, prenant les valeurs « 0 » ou « 1 », symboles de base manipulés par cette machine de Turing sophistiquée qu'est l'ordinateur.

Notons que les bits sont représentés par des chiffres - permettant de former des nombres *en base 2* - ce qui a conduit longtemps à considérer, de façon réductrice, l'ordinateur comme une puissante machine à calculer. C'est d'ailleurs ce que signifie la racine latine *comput* - de *Computus*, *Calcul* - sur laquelle est construit le terme anglo-saxon *computer*.

Pourtant, le 0 et le 1 peuvent tout autant signifier « faux » et « vrai ». Le cœur de tous les microprocesseurs est constitué d'une multitude de portes logiques - on les chiffre en milliards dans les ordinateurs actuels - qui ne font en définitive rien de plus qu'effectuer des traitements logiques élémentaires « ET », « OU », et « NON », selon des principes qui trouvent leur origine dans la logique aristotélicienne et qui seront formalisés dans l'algèbre de Boole au XIXème siècle. Autrement dit, *autant que des calculs, un ordinateur effectue des traitements logiques, et possède donc une capacité de déduction*.

Machine à calcul ou machine logique ? C'est à l'utilisateur d'en décider, et peut-être est-ce là le fond de la question philosophique soulevée par l'IA : l'ordinateur est fondamentalement une machine de Turing qui se contente de manipuler des symboles abstraits, *comme des choses en soi vides de toute signification auxquelles seul l'esprit humain attribue un sens*.

Dès 1905, soit près de trente ans avant les travaux de Turing, le grand mathématicien Henri Poincaré formulait cette idée de façon prémonitoire, et avec un certain humour, imaginant une machine logique capable d'effectuer des raisonnements mathématiques « où l'on introduirait les axiomes par un bout pendant qu'on recueillerait les théorèmes à l'autre bout, comme cette machine légendaire de Chicago où les porcs entrent vivants et d'où ils ressortent transformés en jambons et en saucisses.² »

Des débuts prometteurs

Dans les années 50, le film de science-fiction *Forbidden Planet* met en scène *Robby* le robot. Comme son cousin le « droïde de protocole » C3PO de *Star Wars* apparu sur les écrans vingt ans plus tard, c'est l'archétype de la machine sympathique, capable de communiquer par la parole avec les humains, voire même d'exprimer quelques émotions.

C'est à cette même époque, en 1957, que le linguiste Noam Chomsky publie sa thèse³, décrivant les *grammaires génératives*, ou *grammaires formelles* : en substance, celles-ci consistent à modéliser une langue en un ensemble d'axiomes d'une part (les catégories syntaxiques telles que le verbe, le nom, l'article, etc., ainsi que tous les mots du dictionnaire), et de règles syntaxiques d'autre part, qui ne sont rien de plus que les règles de grammaire qu'on apprend à l'école. Il sera ainsi possible, en appliquant ces règles aux axiomes, de déduire toutes les phrases grammaticalement correctes de la langue.

Sous cet angle, il existe une analogie parfaite entre la structure d'une langue et une théorie mathématique. Celui qui parle ne se contente donc pas de faire de la prose sans le savoir, *il déduit aussi des « théorèmes » à partir d'« axiomes »*. A l'inverse, s'il reconnaît une phase écrite dans sa langue (soulignons qu'il n'est pas ici question du *sens* de cette phase, mais seulement de sa forme grammaticale), *il effectue mentalement l'équivalent de la démonstration d'un théorème !*

On pourrait dire qu'à la fin des années 1950, tous les concepts sont réunis pour concevoir *Robby* le robot parlant, et des experts sans doute un peu trop optimistes prédisent à l'époque son apparition pour la fin de la décennie suivante. Cela prendra en réalité plus de 60 ans. Ce n'est qu'aujourd'hui, avec les *assistants personnels intelligents*, qu'une machine grand public peut engager un semblant de dialogue avec des humains - et encore, pour étonnante qu'elle soit sur le plan technologique, cette communication trouve-t-elle très vite ses limites ...

Si la recherche fondamentale en IA se poursuit dans les années 1960, elle ne trouve pas encore de véritable application dans l'industrie. Celle-ci s'intéresse alors prioritairement aux deux capacités premières de l'ordinateur : gérer de l'information dans des bases de données, avec le développement des « Business Machines », et effectuer des calculs mathématiques complexes, avec au premier rang des applications dans les domaines scientifique ou militaire.

Il faudra attendre les années 1970 pour que se produise une nouvelle avancée majeure, permise par l'apparition de nouveaux langages de programmation. Ceux-ci - d'ailleurs souvent basés sur des approches linguistiques - donneront naissance à la *Programmation Orientée Objet* » (POO), aujourd'hui d'un usage généralisé : alors que les premiers programmes informatiques étaient composés de lignes numérotées, interconnectées par des instructions « GoTo » ou « GoSub » qui rendaient leur maintenance difficile et source de bugs multiples, la POO offre un mode de programmation plus souple - on pourrait même dire élégant - ouvrant plus facilement la voie à la programmation de logiciels d'Intelligence Artificielle.

Déceptions des années 1980

Les *Systèmes Experts* se développent durant les années 1980. Ces logiciels, visant notamment le tout nouveau marché des micro-ordinateurs personnels, sont conçus pour proposer des *outils d'aide à la décision*. Ils reposent sur des principes de logique formelle, dont l'exemple le plus élémentaire est le syllogisme.

Tous les syllogismes, en effet, n'ont pas la transparence de celui par lequel on démontre que Socrate est mortel ... Au XIXème siècle, Lewis Caroll s'efforçait de trouver de tels exemples de déductions vraies sans être évidentes :

« *Tous les lapins non gourmands sont noirs ;
Aucun vieux lapin n'est exempt de gourmandise.
Donc :
Certains lapins noirs ne sont pas vieux.* »⁴

Plus sérieusement, un système expert est capable de gérer un grand nombre de propositions, manipulées par des règles de déduction susceptibles de conduire à des conclusions non-intuitives. Une application possible est l'*aide au diagnostic*, chaque proposition portant par exemple sur la description d'un symptôme, et la conclusion caractérisant une pathologie. Le système expert ne se contentera pas d'établir un diagnostic, il le fera en ne posant au patient que le minimum de questions nécessaires.

Pourtant, malgré quelques réussites notables, les systèmes experts n'ont pas su répondre aux attentes qu'ils avaient suscitées, au point même de provoquer une remise en cause, aux yeux de certains observateurs, de l'intérêt pratique de l'Intelligence Artificielle.

La puissance des ordinateurs de l'époque - *a fortiori* des micro-ordinateurs - était probablement insuffisante pour effectuer dans de bonnes conditions ces traitements complexes. Plus fondamentalement, sur le plan théorique, pouvait-on aussi s'interroger sur la difficulté - *voire dans certains cas l'impossibilité* ? - à retranscrire explicitement, dans toute sa subtilité, le savoir-faire d'un expert en un ensemble de règles de déduction.

Nouvel essor des années 1990

En décembre 1991, le *High Performance Computing Act*⁵, adopté par le Congrès des États-Unis, allait marquer l'essor des *autoroutes de l'information*, autrement dit *internet*. Ce réseau avait été créé plus de vingt ans auparavant, mais était resté confiné jusqu'alors - hors applications de défense - essentiellement dans le milieu universitaire et dans quelques entreprises de pointe.

Internet va avoir deux incidences sur le développement de l'IA : tout d'abord, la capacité d'interconnecter des ordinateurs, et donc d'augmenter potentiellement leur capacité globale de traitement. Mais aussi - et surtout - de faciliter l'accès à des quantités considérables de données - la *data*, qui conduira plus tard au *big data* - nécessaires à l'apprentissage de nouveaux algorithmes.

Dans le même temps, la puissance des micro-processeurs se développe, et des logiciels toujours plus performants conduisent notamment, de façon spectaculaire, le super-ordinateur *Deep Blue* à battre en 1997 le champion du monde d'échec Garry Kasparov.

Mais une autre révolution se prépare, qui prendra véritablement son essor dans les années 2000, avec les *réseaux neuronaux*.

Succès des réseaux neuronaux

Les réseaux de neurones ne sont pas nouveaux. Les premières publications scientifiques sur le sujet datent de 1950, et sont inspirées tout naturellement par l'observation du neurone biologique - même s'il s'agit là d'une métaphore qui n'a probablement que peu à voir avec la complexité de cette cellule.

Le neurone de l'intelligence artificielle, dans son principe, n'a rien de complexe : son rôle se borne à recueillir des données qu'il va traiter par des fonctions mathématiques élémentaires (certaines sont de simples combinaisons linéaires) pour produire de nouvelles données ... Celles-ci, produites par un premier ensemble de neurones - une première *couche* - peuvent à leur tour être utilisées comme source pour une seconde couche, et ainsi de suite... Le réseau est alors constitué de multiples niveaux, de plus en plus performants et *profonds*, notion qui a conduit au terme actuel de *Deep Learning*.

L'intérêt pratique de ce réseau est la simplicité de mise en œuvre de sa phase d'apprentissage : préalablement alimenté par une très grande quantité de données, des dizaines de milliers de photos de chats par exemple, il intégrera dans ses neurones l'information utile pour détecter, sur tout nouveau cliché qui lui est soumis, la présence de l'animal ...

Contrairement aux systèmes experts, aucune compréhension du sujet n'est ici nécessaire : un programmeur humain n'aura pas à traduire le concept « chat » en d'hypothétiques règles logiques capables d'analyser le contenu d'une photo. Seule compte la quantité de données brutes utilisées lors de l'apprentissage.

Il faut cependant admettre que les réseaux de neurones, compte tenu de la fiabilité insuffisante de leurs premiers résultats, sont longtemps restés dans les laboratoires de recherche. Un logiciel capable de reconnaître une adresse postale manuscrite avec une fiabilité de 80 ou 90 % n'aurait qu'une utilité relative. Pour être opérationnel, il lui faut, au minimum, atteindre un seuil de 99,9 %...

Cet objectif deviendra accessible dans les années 2000. Parallèlement aux micro-processeurs (le *Central Processing Unit*, ou CPU), on exploite alors plus systématiquement les ressources de l'autre puce d'un micro-ordinateur, *a priori* destinée à gérer son interface graphique (le *Graphic Processing Unit*, ou GPU), dont on prend conscience que le mode de calcul est particulièrement bien adapté aux traitements neuronaux. Et à partir de 2005 apparaissent sur Internet des bases d'images référencées - de vastes collections de photographies d'animaux et d'objets de toutes sortes... -, disponibles en libre accès, et destinées à alimenter l'apprentissage de nouveaux algorithmes.

La base la plus célèbre est *ImageNet*, présentée en 2009 par l'université de Princeton, comptant plusieurs millions de photographies. Lors de la compétition ILSVRC (*ImageNet Large Scale Visual Recognition Competition*), où s'affrontent chaque année les algorithmes de reconnaissance d'images exploitant cette base, les meilleurs taux de réussite seront de 99,5 % en 2010, puis de 99,7% en 2011. *Ils passeront la barre des 99,9% en 2014...*

Aujourd'hui les réseaux neuronaux s'imposent, au point qu'on les assimile souvent - à tort - à l'Intelligence Artificielle. Les smartphones de dernière génération intègrent des circuits entièrement dédiés aux calculs neuronaux, utilisés pour déverrouiller l'appareil par reconnaissance faciale. En 2015, *AlphaGo* est le premier programme informatique - basé sur des algorithmes neuronaux - capable de battre un joueur humain au jeu de Go. En 2017, une variante de ce programme, *AlphaZero*, dépasse le niveau des meilleurs logiciels de jeu d'échec, avec - contrairement à son prédecesseur *Deep Blue* - pour seules et uniques données de base les règles du jeu, son expérience étant acquise en jouant contre lui-même trois jours durant un nombre incalculable de parties.

Conclusion : quelques questions soulevées par l'IA

Avec de tels outils, l'IA permet désormais de concevoir des machines suffisamment puissantes pour intervenir, de façon directe ou indirecte, dans des processus de décision. Cela pose, par nature, des questions d'ordre éthique : de quel *droit* une machine peut-elle intervenir dans un tel processus ?

Les interrogations soulevées par l'usage des machines ne sont pas nouvelles, et constituent même depuis toujours un thème majeur de la science-fiction. Excepté qu'il s'agit aujourd'hui d'une réalité : *il est essentiel de « garder la main » sur la puissance de l'IA*, comme le rappelleront en 2015 des personnalités du monde scientifique ou de grandes entreprises technologiques, telles que le physicien Stephen Hawking ou l'ingénieur-entrepreneur Elon Musk, dans une *lettre ouverte*⁶ mettant en garde contre certains risques de l'IA.

Ces questions ne doivent pas être considérées comme un réquisitoire contre l'IA. Mais il serait irresponsable de les ignorer. On peut en relever quelques-unes :

- **L'IA produit-elle des boîtes noires ?** Est-il possible de justifier le résultat d'un traitement complexe, autrement-dit de reconstituer de façon précise et compréhensible tout le cheminement qui a conduit à sa production ? Contrairement aux systèmes basés sur des inférences logiques, il est difficile de reconstituer le mode de fonctionnement interne d'un réseau de neurones. Au-delà de l'efficacité

opérationnelle avérée de ces systèmes, des progrès scientifiques sont à faire pour mieux comprendre et maîtriser leur développement.

- **Quel est le statut de la data exploitée par l'IA ?** Le citoyen est-t-il véritablement conscient de l'usage qui peut être fait de la *data* qu'il produit et qui alimente des algorithmes d'IA ? Cette *data* est-elle du domaine de la vie privée, que la loi a vocation à protéger (en Europe, le *Règlement Général sur la Protection des Données*, ou RGPD, tente d'aller dans ce sens) ? Ou, selon une approche plus anglo-saxonne, l'individu est-il le seul propriétaire de ses données et donc libre d'en disposer, y compris en les cédant s'il le souhaite à un opérateur privé pour bénéficier, dans une logique commerciale, d'un service gratuit ou à prix réduit ?
- **L'IA ne réduit-elle pas la réalité à des phénomènes quantifiables ?** La *data* qui alimente un logiciel d'IA ne risque-t-elle pas de limiter la perception de chaque domaine qu'elle aborde à des mesures chiffrées : l'évaluation de la qualité d'un service à une simple note de 0 à 10 attribuée par des consommateurs, la santé à un ensemble de mesures biométriques effectuée par un bracelet connecté, l'intelligence à un Quotient Intellectuel, ou l'individu à un « crédit social » actualisé par un système de reconnaissance faciale détectant dans la cité ses supposées « incivilités » ... ? Et, au-delà, ces données une fois massivement récoltées et traitées, ne risquent-elles pas d'imposer leur moyenne comme une norme, écartant parce qu'« a-normaux » les cas atypiques ou simplement originaux ?
- **Quelle place souhaite-t-on donner à l'IA dans la société ?** L'IA est - et sera de plus en plus - en capacité d'effectuer des tâches jusqu'alors assurées par des humains, et donc de supprimer des emplois, y compris des emplois qualifiés. C'est un choix de société : jusqu'à quel point souhaite-t-on effectuer cette substitution ?
- **Peut-on anticiper les développements futurs de l'IA ?** La recherche en Intelligence Artificielle n'a pas de raison de s'arrêter, et, succédant à l'IA dite *faible* d'aujourd'hui, c'est-à-dire ne pouvant se confondre - malgré son efficacité sur certaines tâches - avec un esprit humain, on peut s'interroger sur l'hypothèse de l'émergence, demain ou après-demain, d'une *IA forte* aux capacités bien plus élevées. Une telle IA est-elle concevable, et même souhaitable ?

Toutes ces questions ne portent pas tant sur la nature en elle-même de l'Intelligence Artificielle, que sur l'usage qui peut en être fait. En ce sens, elles rejoignent les interrogations qui ont toujours accompagnées, à juste titre, l'émergence de nouvelles technologies. Ces réflexions critiques doivent se poursuivre de façon constructive et responsable, sans jamais perdre de vue les bénéfices apportés par une IA maîtrisée, qui sera d'une aide précieuse - pour ne pas dire nécessaire - face aux défis majeurs du XXI^e siècle.

¹ Poe E.A., traduction française Baudelaire C. Histoires extraordinaires. Nouvelles histoires extraordinaires. Michel Lévy frères, Paris 1856.

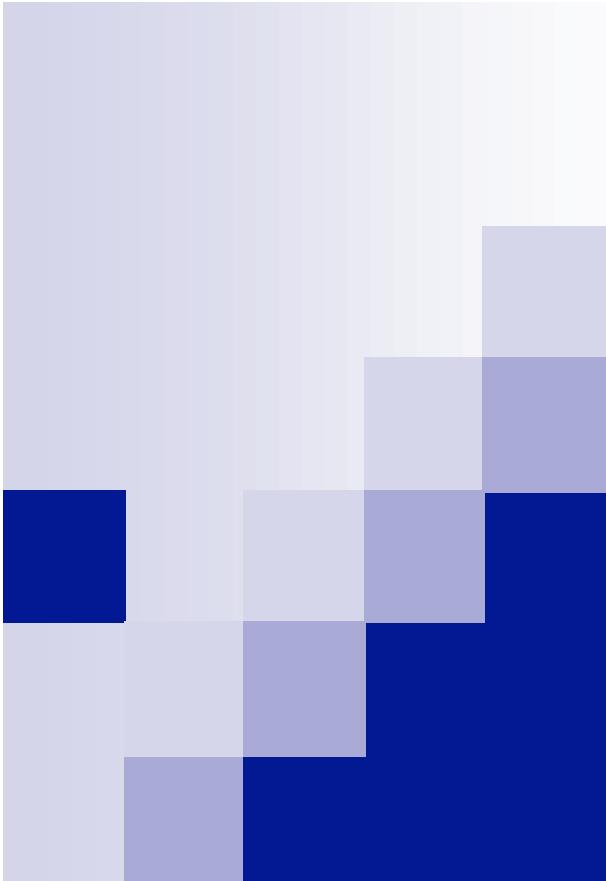
² Poincaré H. Science et Méthode, Bibliothèque de philosophie scientifique, Flammarion, 1908.

³ Chomsky N. Syntactic Structures. Mouton & CO. 1957.

⁴ Carroll L. Symbolic Logic. London Macmillan ed. 1896.

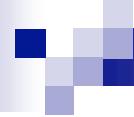
⁵ US Congress. High-Performance Computing Act of 1991. <https://www.congress.gov/bill/102nd-congress/senate-bill/272>

⁶ Texte collectif. Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. 2015. En ligne sur le site Future of Life : <https://futureoflife.org/ai-open-letter>



Intelligence Artificielle





Plan du cours

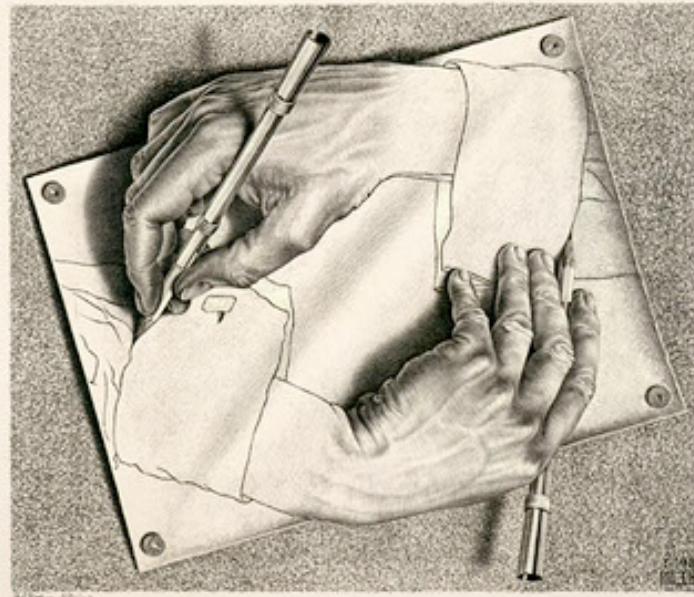
- Champ de l'IA
- Historique
- Systèmes Formels
- Langage Prolog : Notions de Base
- Systèmes Experts
- Constraints Satisfaction Problems



Champ de l'IA

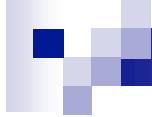


Champ de l'Intelligence Artificielle



Peut-on penser la pensée ?





Champ de l'Intelligence Artificielle

Quelques objectifs ...

- réagir avec discernement à des situations nouvelles,
- tirer profit de circonstances fortuites,
- discerner le sens de messages ambigus ou contradictoires,
- juger de l'importance relative de différents éléments d'une situation,
- trouver des similitudes entre des situations malgré leurs différences,
- établir des distinctions entre des situations malgré leurs similitudes,
- synthétiser de nouveaux concepts malgré leurs différences,
- trouver de nouvelles idées,
-



Champ de l'Intelligence Artificielle

... quelques questions ...

- Lorsqu'il formule un nouveau théorème, le mathématicien est-il un découvreur, ou un inventeur ?
- Un théorème est-il démontré lorsque seule la puissance de calcul d'un ordinateur permet de le démontrer ? (exemple des premières démonstrations du théorème des quatre couleurs)
- Peut-on concevoir une machine capable de produire mécaniquement tous les livres possibles ? (voir la *Bibliothèque de Babel* de Borges).
- Quel est le rapport entre la forme de symboles (des mots, des phrases, des symboles mathématiques, ...) et le sens qu'ils expriment ?
- A-t-on besoin de connaître le sens de symboles pour les manipuler ? (voir la machine à fabriquer des théorèmes de Poincaré).
- Existe-t-il un critère objectif permettant de décider si un système est ou non intelligent ?
- ...



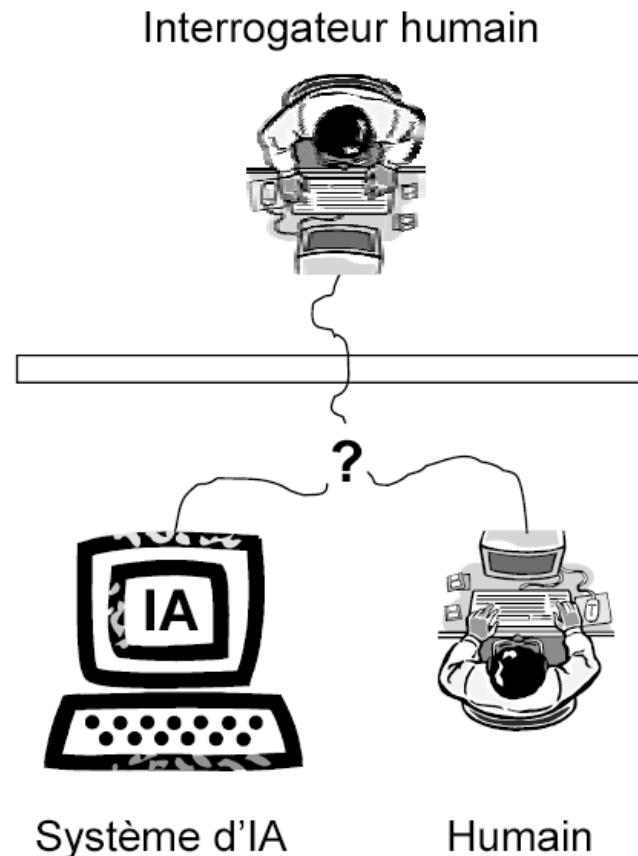
Champ de l'Intelligence Artificielle

... et quelques définitions :

- « *Si l'informatique est la science du traitement de l'information, l'I.A. s'intéresse à tous les cas où ce traitement ne peut être ramené à une méthode simple, précise, algorithmique* » (Laurière 87).
- *L'IA est « l'application de logiciels et de techniques de programmation mettant en lumière les principes de l'intelligence en général, et de la pensée humaine en particulier »* (Boden).
- ... « *l'étude de l'intelligence, indépendamment de sa manifestation chez l'homme, l'animal ou la machine* » (McCarthy)
- ... « *la poursuite de la métaphysique par d'autres moyens* » ! Longuet-Higgins.
- ... « *la science s'intéressant aux machines qui réalisent ce qui, fait par l'homme, nécessiterait de l'intelligence* » (Minsky).
- "L'intelligence artificielle peut être définie comme la tentative d'obtenir des machines comme celles qu'on voit dans les films." ..(Russell Beale)

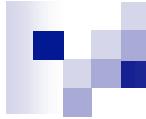


Champ de l'Intelligence Artificielle



Le test de Türing ...





Champ de l'Intelligence Artificielle

	Penser comme des humains	Penser rationnellement
Pensée	<p>« The exciting new effort to make computers think ... <i>machines with minds</i>, in the full and litteral sense » (Haugeland, 1985)</p> <p>« [The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ... » (Bellman, 1978)</p>	<p>« The study of mental faculties through the use of computational models » (Charniak and McDermott, 1985)</p> <p>« The study of computations that make it possible to perceive, reason, and act » (Winston, 1992)</p>
Action	Agir comme des humains	Agir rationnellement
	<p>« The art of creating machines that perform functions that require intelligence when performed by people » (Kurzweil, 1990)</p> <p>« The study of how to make computers do things at witch, at the moment, people are better » (Rich and Knight, 1991)</p>	<p>« Computational Intelligence is the study of the design of intelligent agents » (Poole <i>et al.</i>, 1998)</p> <p>« AI ... is concerned with intelligent behavior in artifacts » (Nilsson, 1998)</p>



Champ de l'Intelligence Artificielle - Break

Deux exemples - problème de la reconnaissance de forme

Pourquoi peut-on lire aisément le paragraphe suivant ?

Sleon une édtue de l'Uvinertisé de Cmabrigde, l'odrre des ltteers dnas un mtos n'a pas d'ipmrotncae, la suele coshe ipmrotnate est que la pmeirère et la drenière soit à la bnnoe pclae. Le rsete peut êrte dnas un dsérorde ttoal et vuos puoevz tujoruos lrie snas porblème. C'est prace que le creaveu hmauin ne lit pas chuaqe ltetra elle-mmée, mias le mot cmome un tuot.

Combien y-a-t-il de "F" dans cette phrase ?

+++++
FINISHED FILES ARE THE RE-
SULT OF YEARS OF SCIENTIF-
IC STUDY COMBINED WITH THE
EXPERIENCE OF YEARS
+++++

Réponse : non il n'y en a pas 3 mais 6 ! (le cerveau humain a tendance à oublier le « F » de « OF » ...)



Historique



Historique - Quelques dates

- **Jusqu'à la première moitié du XXième siècle** : il s'agit avant tout de construire les bases philosophiques et mathématiques sur lesquelles pourra se développer l'IA.
- **A noter, quelques projets de « machines à penser », qui sont essentiellement des « machines à calculer »** : projet de machine à calculer de Pascal au XVIIème siècle, projet « d'automate rationnel » de Leibnitz au début du XVIII ème, machine de Babbage au XXième, ...
- **Années 1940 / 50 : naissance de l'IA.** Modèles mathématiques de Pitts et MacColloch en 1943; machine de Turing en 1950; théorie de l'information de Shannon,...
- **Années 50 - 60 : essor de l'IA** : conférence de Darmouth en 1956 consacrant le terme d'Intelligence Artificielle; développement des ordinateurs, des langages informatiques de type Lisp, ...



Historique - Quelques dates

- **Années 70 : nouveaux modèles de représentation et de traitement des connaissances.** Systèmes experts, création du langage Prolog orienté vers le traitement de la langue naturelle en France au début des années 70, ...
- **Années 80 : phase d'industrialisation :** diffusion d'applications industrielles, commercialisation de progiciels, ...
- **Depuis les années 90 : retour à des projets plus réalistes** (on parle moins des systèmes experts par ex.). Nouveaux concepts (ex. Agents intelligents), développement des applications liées à la recherche de l'information (parallèlement au développement d'Internet), résurgence des concepts de réseaux neuronaux et d'algorithmes génétiques, ...



Historique - Une référence : la machine de Türing

La machine de Türing (années 1940-50) est un automate théorique disposant :

- d'une source d'information, matérialisée par un ruban (théoriquement infini), sur lequel sont imprimés des symboles ($X_1, X_2, \dots X_n$).
- d'un "scanner-marqueur" capable de lire et d'écrire des symboles sur le ruban,
- d'un mécanisme possédant un nombre fini d'états internes notés $q_1, q_2, \dots q_p$.

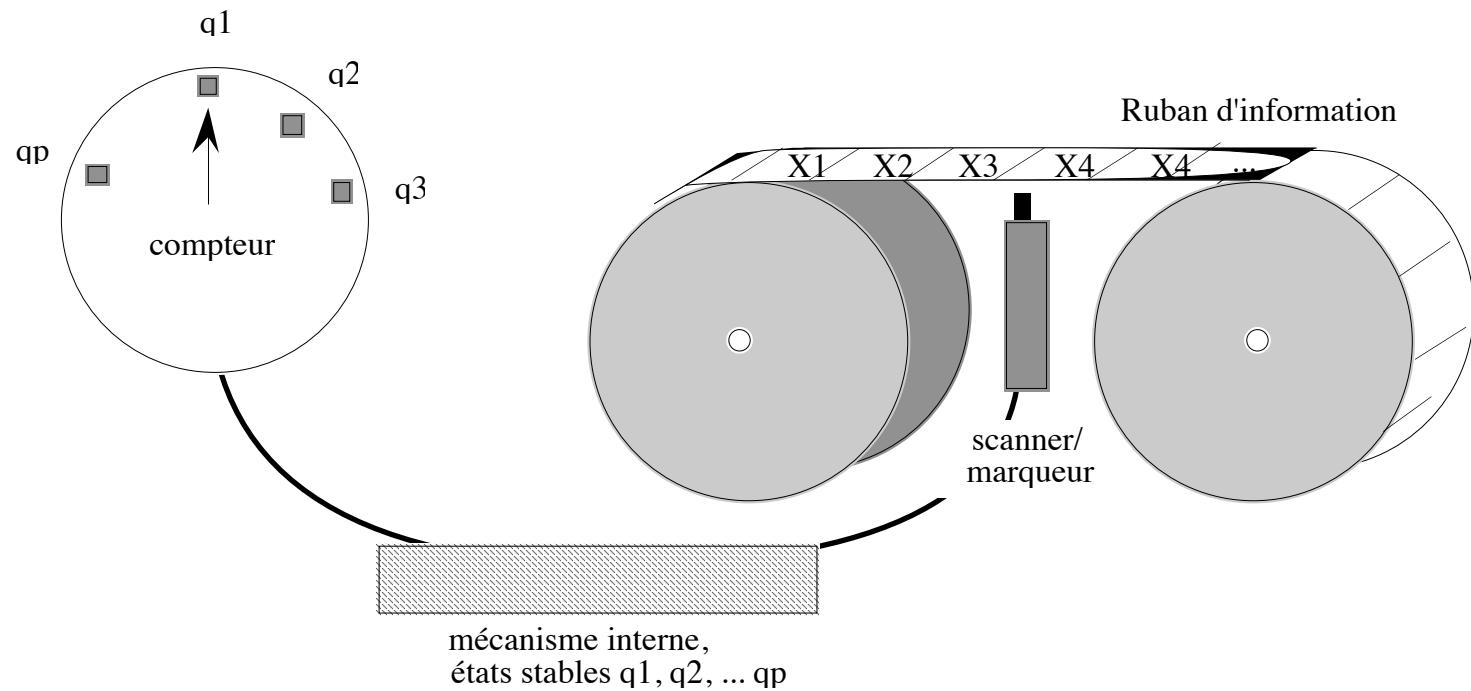
Ce mécanisme affichera son état courant sur un compteur.

Connaissant l'état interne q_i du mécanisme, et le symbole X_j pointé sur la bande par le scanner, la machine pourra effectuer des traitements élémentaires :

- en réécrivant, ou non, un nouveau symbole sur le ruban,
- puis en déroulant, ou non, le ruban d'un cran vers la droite ou vers la gauche,
- et en modifiant, ou non, l'état interne q_i du mécanisme.



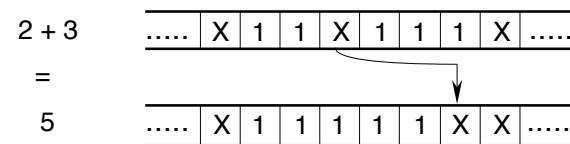
Historique - Une référence : la machine de Turing



Historique - Une référence : la machine de Tûring

Exemple de machine de Tûring : machine à additionner.

Dans cet exemple, tout nombre sera codé sur le ruban par une série de "1" (3 sera codé "111", 4 par "1111", etc ...), chacun de ces nombres étant délimité par un séparateur "X". Une addition telle que « 2+3=5 » pourra donc être représentée par :



Pour effectuer cette opération, la machine de Tûring doit posséder au moins trois états internes q_1 , q_2 , et q_3 , et appliquer les règles suivantes :

	Situation de départ		Situation à l'étape suivante		
	Etat interne du mécanisme	Symbole lu par le scanner	Réécriture du symbole	Décallage du ruban	Etat interne du mécanisme
1	q_1	1	1	gauche	q_1
2	q_1	X	1	gauche	q_2
3	q_2	1	1	gauche	q_2
4	q_2	X	X	droite	q_3
5	q_3	1	X	stop	q_3
6	q_3	X	X	stop	q_3



Systèmes formels



Systèmes formels - La Bibliothèque de Babel ...

"La Bibliothèque de Babel est une sphère dont le centre véritable est un hexagone quelconque, et dont la circonférence est inaccessible /.../. Il n'y a pas, dans la vaste Bibliothèque, deux livres identiques. De ces prémisses incontournables, il déduit que la Bibliothèque est totale, et que ses étagères consignent toutes les combinaisons possibles des vingt et quelques symboles orthographiques (nombre, quoique très vaste, non infini), c'est-à-dire tout ce qu'il est possible d'exprimer, dans toutes les langues. Tout : l'histoire minutieuse de l'avenir, les autobiographies des archanges, le catalogue de la Bibliothèque, des milliers et des milliers de catalogues mensongers, la démonstration de la fausseté de ces catalogues, la démonstration de la fausseté du catalogue véritable /.../. »

J.L. Borges - La Bibliothèque de Babel - Fictions - 1941.



Systèmes formels - Définition

Un système formel est « un ensemble de données purement abstrait, sans lien avec l'extérieur, qui décrit les règles de manipulation d'un ensemble de symboles traités de façon uniquement syntaxique, c'est-à-dire sans considération de sens (sémantique).

Il est constitué :

- 1°/ d'un alphabet fini de symboles;
- 2°/ d'un procédé de construction des mots du système formel;
- 3°/ d'un ensemble d'axiomes qui sont des mots;
- 4°/ d'un ensemble fini de règles de déduction qui permettent de déduire d'un ensemble fini de mots un autre ensemble de mots.

Elles sont de la forme :

$U_1 \text{ et } U_2 \text{ et } \dots \text{ et } U_p \Rightarrow W_1 \text{ et } W_2 \text{ et } \dots \text{ et } W_n$
où les U_i et les W_j sont des mots du système formel.
La flèche " \Rightarrow " se lit "permet de déduire". »

(Laurière 87)

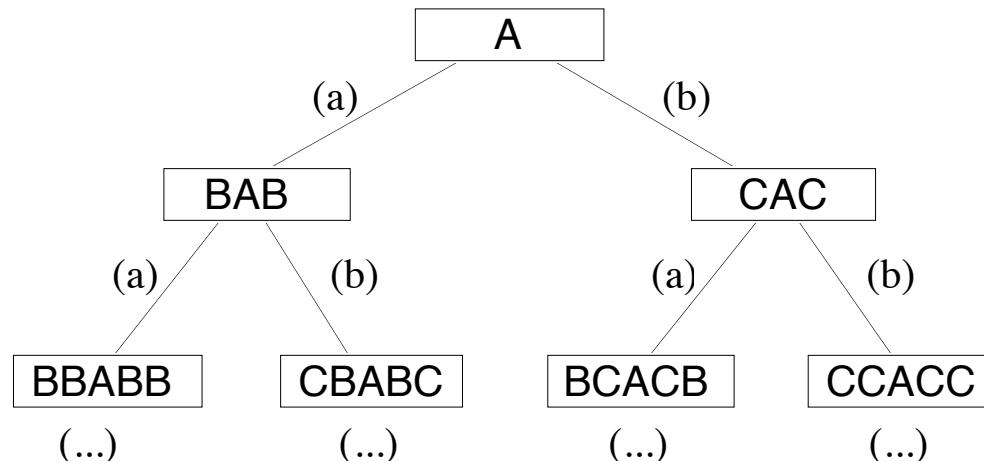


Systèmes formels - Exemple

Exemple du système « BAB »:

- 1°/ alphabet = les trois symboles A, B et C,
- 2°/ mots = toute suite finie de symboles de l'alphabet. Par exemple ""(chaîne vide), "A", "AAAAAB", "BABBCA", etc ...
- 3°/ axiome unique : "A"
- 4°/ règles de déduction : soient n et m deux mots quelconques,
 - a) $mAn \Rightarrow BmAnB$
 - b) $mAn \Rightarrow CmAnC$

Ceci peut se traduire par un graphe, aussi appelé *arbre de dérivation* :



Systèmes formels - Décidabilité

On dira qu'un système formel est **décidable** s'il est possible, en un nombre fini d'étapes, de déterminer si un mot quelconque peut être ou non déduit des axiomes du système.

Exemple : le système « BAB » est-il décidable ? Autrement dit, soit un mot donné, par exemple « CCABBAC », peut-on dire s'il appartient, ou non, à l'ensemble des mots produits par le système « BAB » ?

Il existe plusieurs façons de répondre à cette question :

- soit en parcourant l'arbre de dérivation de l'axiome « A » vers les théorèmes, et en vérifiant si « CCABBAC » est un de ces théorèmes. On verra qu'il s'agit là d'une stratégie de recherche dite « en chainage avant ».
- soit en tentant d'appliquer les règles de déduction en sens inverse, pour vérifier s'il est possible, depuis « CCABBAC » de « remonter » jusqu'au théorème « A ». On verra qu'il s'agit d'une stratégie de recherche dite « en chainage arrière ».
- il existe une troisième solution : constater que tous les théorèmes produits par le système « BAB » sont de la forme *au centre la lettre A, et de part et d'autre une combinaison symétrique de B et de C*.



Systèmes formels - Exemple du jeu d'Echecs



Olivier Boisard - Cours d'Intelligence Artificielle



Systèmes formels - Exemple du jeu d'Echecs

L'ensemble des parties du jeu d'échecs pourrait ainsi être représenté par le système formel comprenant :

- **Un alphabet** : tout symbole représentant une pièce du jeu positionnée sur l'échiquier (toutes les pièces possibles, sur toutes les cases possibles). Exemple: "Fou Noir en A1", " Cavalier Blanc en B5", etc ...
- **Des mots** : toute combinaison finie de pièces positionnées. Toute configuration possible de l'échiquier peut donc être représentée par un mot (les configurations "absurdes" comprenant par exemple plusieurs pièces sur la même case sont également des mots),
- **Un axiome** : mot représentant la configuration initiale de l'échiquier.
- **Des règles de déduction** : les règles de mouvement des pièces sur l'échiquier. Autrement dit, comment partant d'une configuration du jeu (donc d'un mot) déplacer une pièce (donc créer un autre mot).



Systèmes formels - Exemple du jeu d'Echecs

L'arbre de dérivation présentera **toutes** les parties possibles du jeu, se terminant (extrémité de chaque branche de l'arbre), soit par un « mat », soit par un « pat ».

Or aucune machine n'a une mémoire suffisamment puissante pour mémoriser l'intégralité de cet arbre...

L'automate joueur d'échec doit donc s'"adapter" à toute configuration nouvelle de l'échiquier, pour "inventer" les réponses les plus appropriées. Il anticipera, dans la limite de sa mémoire, n coups à l'avance, en simulant alternativement

- le mouvement qui lui est le plus favorable, lorsque c'est à lui de jouer,
- le mouvement le plus défavorable, lorsque c'est au tour de l'adversaire

D'où le nom de cette méthode, dite "min-max". Il choisira en définitive la déplacement qui, au nième coup anticipé, est susceptible de l'amener aux situations "globalement les plus favorables".

Pour **évaluer** les situations *favorables* ou *défavorables*, autrement dit limiter l'exploration de l'arborescence en éliminant les branches "sans intérêt", l'automate utilisera *des principes sortant du cadre explicite défini par les seules règles du jeu d'échecs, et s'appuyant sur la pratique d'un expert.*

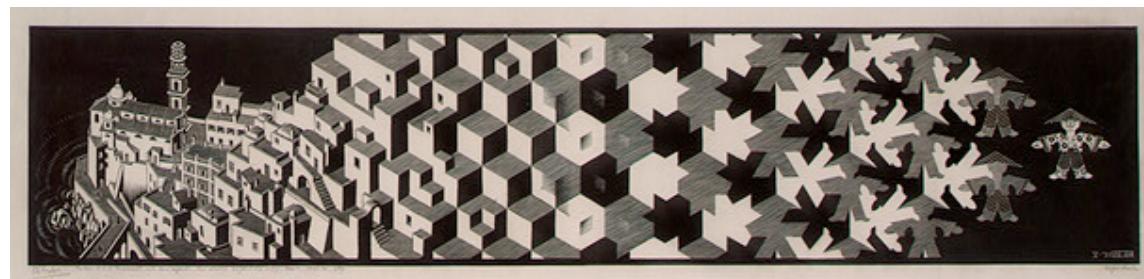




Systèmes formels

« Ainsi, c'est bien entendu, pour démontrer un théorème, il n'est pas nécessaire ni même utile de savoir ce qu'il veut dire. On pourrait remplacer le géomètre par le piano à raisonner imaginé par Stanley Jevons; ou, si l'on aime mieux, on pourrait imaginer une machine où l'on introduirait les axiomes par un bout pendant qu'on recueillerait les théorèmes à l'autre bout, comme cette machine légendaire de Chicago où les porcs entrent vivants et d'où ils sortent transformés en jambons et en saucisses. Pas plus que ces machines, le mathématicien n'a besoin de comprendre ce qu'il fait. »

*Henri Poincaré, **Science et Méthode**, 1905.*



Langage Prolog



Langage Prolog - Préambule : un mot de Lewis Caroll ...

Soient les propositions suivantes :

- Les animaux sont toujours mortellement offensés si je ne fais pas attention à eux
- Les seuls animaux qui m'appartiennent se trouvent dans ce pré
- Aucun animal ne peut résoudre une devinette s'il n'a reçu une formation convenable dans une école
- Aucun des animaux qui se trouvent dans ce pré n'est un raton-laveur
- Quand un animal est mortellement offensé, il se met toujours à courir en tout sens et à hurler
- Je ne fais jamais attention à un animal qui ne m'appartient pas
- Aucun animal qui a reçu dans une école une formation convenable ne se met à courir en tout sens et à hurler

Que peut-on déduire de l'énoncé : « **cet animal est un raton laveur** » ?

Réponse :

On peut en déduire que cet animal est incapable de résoudre une devinette.

In Lewis CAROLL, La logique sans peine.



Langage Prolog - Rappel : l'ordre de la logique

- Logique d'ordre 0 :

C'est la logique des propositions.

Par exemple, Distance=384_km représente un et un seul mot insécable.

- Logique d'ordre 0+ :

Elle contient des opérateurs de comparaison (ex. $>$, $<$, $=$, \geq , \leq et \neq), des valeurs comparables, des constantes (par exemple, la constante Distance peut être assignée à la valeur 384 km).

- Logique d'ordre 1 :

C'est la logique du calcul des prédictats.

Définition du prédictat : Expression logique dont la valeur peut être vraie ou fausse selon la valeur des arguments, c'est-à-dire une fonction qui renvoie "vrai" ou "faux". La logique d'ordre 1 contient :

- * des variables substituables = variables d'individu,
- * des quantificateurs : Pour tout, Quelque soit, ...



Langage Prolog - Rappel : l'ordre de la logique

Exemples de prédictats en logique d'ordre 1 :

- Tout homme est mortel : $\text{homme}(x) \rightarrow \text{mortel}(x)$ pour tout x ;
- Quelque soit x , $d(x) < 2 \text{ km}$ implique $\text{aller_à_pied}(x)$;
- On peut aussi utiliser des fonctions : $\text{inférieure}(\text{distance}(x), 2 \text{ km})$ implique $\text{aller_à_pied}(x)$.

En logique d'ordre 1, on doit choisir des instances pour les variables substituables : le x de $\text{distance}(x)$.

- Logique d'ordre 2 :

C'est la logique des variables de prédictats.

Elle permet d'écrire des méta-règles, c'est-à-dire des règles portant sur des règles.

* Exemple de méta-règle n°1 : Si R_1 est plus spécifique que R_2 , appliquer R_1 .

* Exemple de méta-règle n°2 : la relation de transitivité :

Quelque soit R {Quelque soit x, y, z ($R(x, y)$ et $R(y, z)$) implique $R(x, z)$ } implique $\text{Transitive}(R)$.

Variable de prédictat : ici, on instancie une règle par une relation.



Langage Prolog - Logique ?

- Cette proposition est fausse (paradoxe d'Epiménide).
- Quel est le plus grand nombre que l'on peut exprimer avec moins de 200 caractères ASCII ? (soit la phrase « le plus grand nombre exprimable avec deux cents caractères, plus un ». Cette phrase fait moins de 200 caractères ...).
- Un ensemble auto-inclusif est un ensemble qui se contient lui même. Exemple : l'ensemble des ensembles; ou l'ensemble de tout sauf les éléphants. Un ensemble qui n'est pas auto-inclusif est dit quelconque. Exemple : ensemble de tous les éléphants (en effet, cet ensemble n'est pas lui-même un éléphant, puisque c'est un ensemble !).
Question : L'ensemble de tous les ensembles quelconques est-il quelconque, ou auto-inclusif ?
- Cette phrase contient quatre fois la consonne "c". Alors :
Celle-ci contient cinq fois la consonne "c". Euh, pardon :
Celle-ci contient six fois la consonne "c". !!!
- On annonce au condamné à mort qu'il sera exécuté dans les dix jours qui viennent, mais que le jour de son exécution sera pour lui une surprise. Ce ne pourra être le jour 10, car étant vivant le 9 il devinera que l'exécution est le 10. Ce ne pourra être le jour 9, car étant vivant le 8 il en déduira que ce sera le 9. Etc. Le condamné en déduit qu'il ne peut être exécuté. Il a été exécuté le 3ième jour, et comme il pensait son exécution impossible par le raisonnement précédent, la condition initiale a bien été respectée.



Langage Prolog - Programmation déclarative

Les langages procéduraux traditionnels , « ont en commun d'être de nature "impérative", c'est à dire qu'ils exigent du programmeur, pour résoudre un problème, de définir selon un algorithme connu de lui-même et de préciser étape par étape la méthode de résolution de ce problème», alors qu'avec un langage orienté objets, « une bonne partie de ces opérations, qui peuvent évidemment être très complexes et très longues, est prise en charge par le système, et non plus par le programmeur qui doit, pour ce qui le concerne, programmer en énonçant des connaissances sur son problème; il convient alors de parler d'une programmation "déclarative", où il revient au programmeur de déclarer "de quoi" est fait le problème, plutôt que "comment" le résoudre » (Bihan 87).



Langage Prolog - Généralités

Le langage **Prolog** (PROgrammation LOGique) est né en France au début des années 70. Il résulte notamment de recherches dans le domaine de la linguistique.

- Un programme Prolog est constitué de clauses. Celles-ci sont de trois type : **faits, règles et questions**.
- Une relation peut être spécifiée par des faits, en énonçant simplement les n-uplets des objets vérifiant la relation, ou en établissant des règles concernant la relation.
- Une procédure est un ensemble de clauses concernant une même relation

Par exemple, le fait élémentaire : **Pierre achète une voiture**

sera représenté par une clause : **acheter (Pierre, Voiture)**

Qu'on peut généraliser par : **acheter (X , Y)**
X et Y étant des objets.



Langage Prolog – Notions de syntaxe Prolog

- Un programme prolog est un ensemble de clauses.
- Chaque clause peut exprimer :
 - un fait élémentaire :
Ex : pere(pierre,paul)
 - des règles :
Ex: parent (X,Y) :- pere(X,Y).
parent (X,Y) :- mere(X,Y).
- Dans l'exemple précédent, la règle « parent » a été définie par deux relations, dont la succession est traitée comme avec un « ou » logique : pour que la règle « parent » soit vérifiée, il faut et il suffit que la première relation soit vérifiée, ou (non exclusif) la seconde.
- Chaque relation se termine par un point « . ».
- Plusieurs conditions, à l'intérieur d'une même clause, sont séparées par une virgule « , », qui a la valeur logique du « et ».
- Les arguments d'une clause sont des individus (des « atomes » : pierre, paul) notés ici en minuscules, ou des variables notées en majuscules représentant un objet quelconque.
- Dans une règle, le symbole « :- » représente le « si » de la logique.
- Lorsque l'interprète prolog est prêt, il affiche « ?- » dans l'attente d'une interrogation.

Exemple : ?- pere(pierre, X). *>question posée après « ?- »*
 X=paul *>réponse de l'interprète Prolog*
 yes *> « oui », il y a une réponse vraie à la question.*

- Pour évaluer une expression arithmétique, on utilisera le prédicat prédéfini « is ».

Exemple : ?- X is 2+5*4
 X= 22
 yes



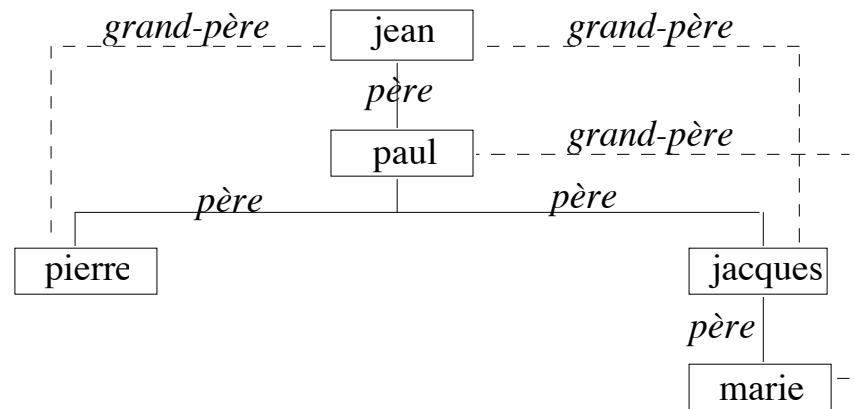
Langage Prolog - Règles

Connaissant, par exemple, les faits élémentaires :

Père (Jean , Paul). (pour "Jean est le père de Paul")
Père (Paul , Pierre).
Père (Paul , Jacques).
Père (Jacques, Marie).

La règle "Grand-Père", permettra de produire (*d'inférer*) de nouveaux faits :

Grand-Père (X , Y) :-
il existe un objet Z avec
Père (X , Z), et
Père (Z , Y) .



Langage Prolog - Questions

Pour l'utilisateur ce codage de l'information présente de nombreux intérêts, notamment celui de pouvoir interroger une "base de connaissances" (qu'on assimilera ici à un ensemble de clauses), aussi bien par des questions "fermées" (appelant des réponses oui/non), que par des questions "ouvertes" à une ou plusieurs inconnues ("trouver les objets pour lesquels cette affirmation est vraie").

Avec la base de faits précédente, l'interrogation :

Père (Jean , Pierre) ?
a pour réponse "faux".

Et l'interrogation :

Grand-Père (Jean , X) ?
(i.e. "Jean est le grand-père de qui ?")
a pour réponse :
X = Pierre, et
Y = Jacques.

On constate que toutes les "connaissances" sont exprimées de la même façon par des clauses. Il n'existe plus de distinction structurelle entre l'information au sens de "process" (celle "qui dit ce qu'il faut faire", le fichier-programme dans un langage traditionnel), et l'information au sens de "data" (celle "sur quoi on travaille", le fichier-données dans un langage traditionnel).



Langage Prolog - Récursivité

Supposons que l'on souhaite compléter la base de connaissances "père/grand-père" par une clause "ancêtre". Une première solution consisterait à définir explicitement toutes les acceptations du mot *ancêtre* :

```
ancêtre(X,Y) :-  
    père(X,Y).   (i.e.: X est l'ancêtre de Y si X est le père de Y ...)  
ancêtre(X,Y) :-  
    père(X,Z),  
    père(Z,Y).   (i.e.: X ancêtre de Y si X grand-père de Y)  
ancêtre(X,Y) :-  
    père(X,Z),  
    père(Z,W),  
    père(W,Y).   (i.e. : X ancêtre de Y si X arrière grand-père de Y)  
etc ...
```

Cette formulation est insuffisante puisqu'elle renvoie à une infinité de règles...

On utilisera donc plutôt un raisonnement récursif du type :

- (a) condition initiale : X ancêtre de Y si X est le père de Y.
- (b) condition récursive : X ancêtre de Y si il existe Z tel que
 - X père de Z et
 - Z ancêtre de Y.

ou encore:

- (a') condition initiale : X ancêtre de Y si X est le père de Y.
- (b') condition récursive : X ancêtre de Y si il existe Z tel que
 - X ancêtre de Z et
 - Z père de Y.



Langage Prolog - Récursivité

Soit en langage Prolog :

Solution (a) (b) :

```
ancêtre(X,Y) if  
    père(X,Y). (a)
```

```
ancêtre(X,Y) if  
    père(X,Z),  
    ancêtre(Z,Y). (b)
```

Solution (a') (b') :

```
ancêtre(X,Y) if  
    père(X,Y). (a')
```

```
ancêtre(X,Y) if  
    ancêtre(X,Z),  
    père(Z,Y). (b')
```



Langage Prolog - Listes

Les *listes* sont utilisées dans Prolog pour manipuler des groupes d'objets. On entendra ici par *liste* un ensemble ordonné d'objets, notés entre crochets et délimités par des virgules.

Exemple :

L = [pierre,paul,jacques].

On notera qu'une liste est *ordonnée*. La liste [pierre,paul,jacques] est donc différente de [jacques,paul,pierre].

On ne dispose que d'un seul opérateur , noté ":" , pour travailler sur une liste : il permet d'isoler la *tête de la liste* (son premier élément), de la *queue de la liste* (la liste constituée des éléments restants).

Exemple :

si : L = [pierre,paul,jacques].
et : L = [X:Y],
alors : X = pierre (un élément),
 Y = [paul,jacques] (une liste).

La liste vide est acceptée, et notée [] .



Langage Prolog - Manipulation de listes

Appartenance à une liste

La clause "appart(X,L)" sera évaluée "vraie" si l'élément X appartient à la liste. Elle sera définie de façon récursive :

- (a) condition initiale : X appartient à la liste L si X est la tête de la liste L,
- (b) récursion : X appartient à la liste L si X est dans la queue de la liste L.

Ceci se traduit par :

```
appart(X,L) :-  
    L=[X:_].           (a)  
appart(X,L) :-  
    L=[_|L1],  
    appart(X,L1)      (b)
```

Dans cet exemple, le symbole "_" est utilisé pour figurer une variable muette, c'est à dire non nécessaire à la suite du traitement. Exemple : dans la condition (a), il n'est pas nécessaire de se préoccuper de la queue de la liste L, dès lors que X est la tête de la liste.



Langage Prolog - Manipulation de listes

On remarquera que la clause "appart" peut être utilisées de deux façons différentes :

- pour vérifier l'appartenance d'un élément à une liste :

```
>appart(pierre,[paul,pierre,jacques]) ?    (i.e. : pierre est-il dans la liste?)
>true                                         (réponse oui)
```

- à l'inverse, pour extraire les éléments d'une liste :

```
>appart(X,[paul,pierre,jaques]) ?          (i.e. : qui est dans la liste ?)
>true                                         (réponse oui)
> X=paul                                     (justification)
> X=pierre
> X=jacques
```

Une question telle que "appart(pierre,L) ?" entraînerait un message d'erreur, puisqu'elle consisterait à demander l'ensemble - infini - des listes contenant l'élément "pierre" ...



Langage Prolog - Manipulation de listes

Concaténation de deux listes :

La clause "concat(L1,L2,L3)" sera évaluée vraie si L3 est la concaténation de L1 et L2.
Exemple : concat([pierre,paul],[jacques,marie],[pierre,paul,jacques,marie]) est vraie.

```
concat([],L2,L3) :-  
    L2=L3.          (condition initiale)  
concat(L1,L2,L3) if  
    L1=[X:L4],  
    concat(L4,L2,L5),  
    L3=[X:L5].      (récursion)
```

A noter : cette clause pourra remplir plusieurs fonctions :

- vérifier la concaténation de trois listes :

> concat([pierre,paul],[jacques,marie],[pierre,paul,jacques,marie]) ?

- vérifier l'élément "manquant" d'une concaténation :

> concat([pierre,marie],L,[pierre,marie,paul]) ?

- extraire toutes les concaténations possibles d'une liste :

> concat(L1,L2,[pierre,paul]) ?



Langage Prolog - Manipulation de listes

Réunion de deux listes :

Il s'agira ici de modifier "concat" pour obtenir une véritable réunion des deux listes, évitant les répétitions.
Autrement dit, obtenir des réponses telles que :

```
> concat([pierre,paul],[marie,pierre], L) ?           (question)
> true                                                 (réponse: oui)
> L = [pierre,paul,marie]                            (justification)
```

La clause "concat" sera modifiée en *ajoutant* une nouvelle condition :

```
concat([],L2,L3) :-  
    L2=L3.                                (condition initiale)  
concat(L1,L2,L3) :-  
    L1=[X:L4],  
    appart(X,L2),  
    concat(L4,L2,L3).                      (= ignorer les doublons)  
                                         (= soit X la tête de L1)  
                                         (= et X déjà dans L2)  
                                         (= alors poursuivre en ignorant X)  
concat(L1,L2,L3) :-  
    L1=[X:L4],  
    concat(L4,L2,L5),  
    L3=[X:L5].                            (récursion)
```

Cette exemple souligne le caractère "*additif*" d'un langage objets. En effet, pour modifier le comportement de la clause "concat", il a suffit d'*ajouter* une condition intermédiaire (ici, *ignorer les doublons*), sans modifier le reste du programme



Langage Prolog - Manipulation de listes

La clause "reverser(L1,L2)" sera évaluée vraie si l'ordre des éléments de L1 est inverse de celui de L2.

Exemple :

```
> renverser([pierre,marie,paul],L) ?      (question)
> true                                     (réponse oui)
> L=[paul,marie,pierre]                   (justification)
```

La clause fera appel à "concat" (dans sa première version, sans élimination de doublons) :

```
renverser (L1,L2) :-  
    L1=[],  
    L2=[].  
renverser(L1,L2) :-  
    L1=[X:L3],  
    renverser(L3,L4),  
    concat(L4,[X],L2).
```

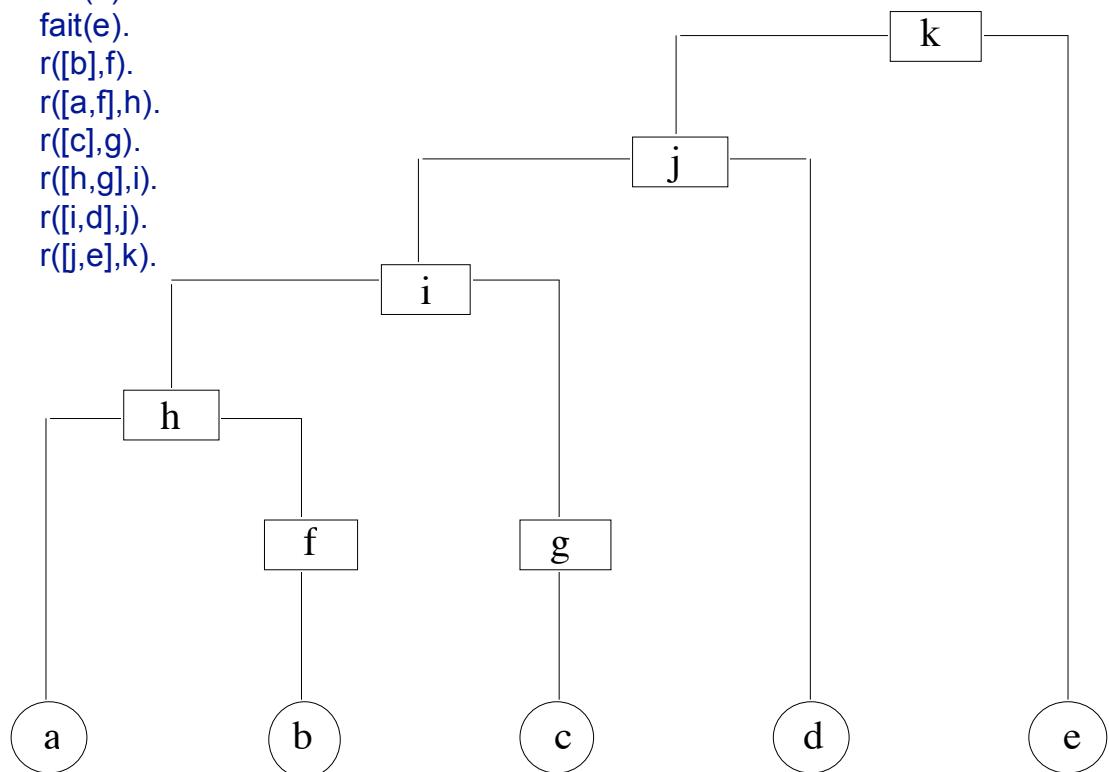
(condition initiale)
(= la liste vide renversée
est la liste vide ...)
(récursion)



Langage Prolog - Stratégies de recherche

Exemple de base de faits :

```
fait(a).  
fait(b).  
fait(c).  
fait(d).  
fait(e).  
r([b],f).  
r([a,f],h).  
r([c],g).  
r([h,g],i).  
r([i,d],j).  
r([j,e],k).
```



Stratégies de recherche
par chainages avant et arrière :

Stratégie 1 :

ar(X) :- *(a/ condition initiale)*
fait(X).
ar(X :- *(b/récursion)*
r(L,X),
appart(Z,L),
ar(Z).

Stratégie 2 :

av(X) :- *(a/ condition initiale)*
fait(X).
av(X) :- *(b/récursion)*
av(Z),
r(L,X),
appart(Z,L).



Langage Prolog – La coupure

La « coupure » est un prédicat prédefini permettant d'agir sur le comportement de l'interprète Prolog lors du retour arrière.

Exemple : soit la relation inserer(X,Xs,Ys), où Xs est une liste triée en ordre croissant d'entiers et Ys est la liste obtenue à partir de Xs en insérant l'entier X.

inserer(X,[],[X]). (1)

inserer(X,[X:Ys],[Y:Zs]) :- X > Y, inserer(X,Ys,Zs). (2)

Inserer(X,[Y:Ys],[X,[Y:Ys]]) :- X=<Y. (3)

Pour résoudre un but inserer(X,Xs,Ys), prolog va essayer ces trois clauses suivant leur ordre d'apparition dans le programme. Si la clause (1) est vérifiée (i.e. si Xs est la liste vide et que Ys est la liste composée de l'unique élément X), il est inutile d'essayer les clauses (2) et (3) lors du retour arrière. On peut « couper » ce processus, avec le prédicat « ! ». Idem si la clause(2) est vérifiée, inutile d'examiner la (3).

inserer(X,[],[X]) :- !. (1)

inserer(X,[X:Ys],[Y:Zs]) :- X > Y,! , inserer(X,Ys,Zs). (2)

Inserer(X,[Y:Ys],[X,[Y:Ys]]) :- X=<Y. (3)



Langage Prolog - Grammaires formelles

On appellera :

- V le vocabulaire, réunion de deux ensembles disjoints :

- V_t , le vocabulaire terminal, c'est-à-dire l'ensemble des symboles qui composent une langue (dictionnaire des tous les mots de la langue),
- V_n , le vocabulaire non-terminal, regroupant l'ensemble des variables, ou catégories syntaxiques, permettant de décrire cette langue.

- un langage sur V_t est un ensemble (éventuellement infini), de chaînes formées par des combinaisons finies de symboles de V_t .

- V^* est l'ensemble des toutes les chaînes finies formées en combinant des symboles terminaux. Un langage est donc un sous-ensemble de V^* .

- une règle de réécriture (ou règle de production) est, au sens large, une relation entre des chaînes formées par des symboles de V . Par exemple, la décomposition d'une phrase (P) en un groupe nominal (GN) et un Groupe Verbal (GV) s'exprimera par la règle :

$P \rightarrow GN + GV.$



Langage Prolog - Grammaires formelles

Un exemple d'application : programmation des grammaires formelles

Les **grammaires formelles** ont pour but d'analyser la langue naturelle en modélisant sa structure syntaxique sous forme de système formel.

C'est **Chomsky**, en 1957, qui dans son ouvrage *Syntactic Structures* , propose le premier modèle de grammaire formelle. "La syntaxe est l'étude des principes et des processus selon lesquels les phrases sont construites dans des langues particulières. L'étude syntaxique d'une langue donnée a pour objet la construction d'une grammaire qui peut être considérée comme une sorte de mécanisme qui produit des phrases de la langue soumise à l'analyse /.../. Le résultat final de ces recherches devrait être une théorie des structures linguistiques où les mécanismes descriptifs utilisés dans les grammaires particulières seraient présentés et étudiés de manière abstraite, sans référence spécifique aux langues particulières " (Chomsky 57).

Les modèles linguistiques de Chomsky reposent sur l'hypothèse d'une décomposition possible de toute phrase en éléments plus simples (mots ou groupes de mots), conformément à des règles explicites



Langage Prolog - Grammaires formelles

On distingue deux types de règles :

- les règles exprimant une relation entre des catégories syntaxiques.

Par exemple : $P \rightarrow GN+GV$.

- les règles de lexicalisation, mettant en relation une catégorie syntaxique et un symbole du vocabulaire terminal V_t . Exemple : $Nom \rightarrow Paul$.

R étant l'ensemble des règles de réécriture, et P le symbole "phrase", on définira une grammaire formelle par le quadruplet :

$$G = (V_n, V_t, R, P).$$

L'exemple suivant est emprunté à Chomsky. Il décrit une grammaire élémentaire basée sur six règles de réécriture :

(I)	P	$\rightarrow GN+GV$
(II)	GN	$\rightarrow Article+Nom$
(III)	GV	$\rightarrow Verbe+GN$
(IV)	$Article$	$\rightarrow the$
(V)	Nom	$\rightarrow man, ball$
(VI)	$Verbe$	$\rightarrow hit, took$

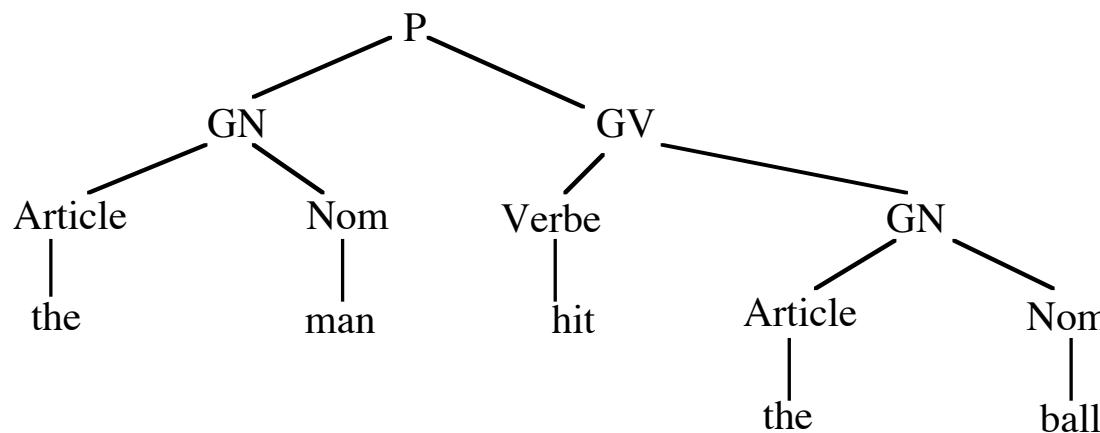


Langage Prolog - Grammaires formelles

Par applications successives de ces règles, on peut aboutir à la décomposition suivante :

P	
GN+GV	(I)
Art+Nom+GN	(II)
Art+Nom+Verbe+GN	(III)
Art+Nom+Verbe+Article+Nom	(II)
the+Nom+Verbe+the+Nom	(IV)
the+man+Verbe+the+ball	(V)
the+man+hit+the+ball	(VI)

Ceci peut se traduire par un graphe, aussi appelé arbre de dérivation :



Langage Prolog - Grammaires formelles

Soit P une phrase (chaîne de caractères) décomposée mot à mot en une liste L. La clause "Phrase(L)" permettra de vérifier si P est ou non une phrase générée par le système, aussi appelée "Expressions Bien Formées" ou EBF.

Phrase (L) :-

 concat(L1,L2,L)
 GN(L1),
 GV(L2).
(L1,L2 sont toutes des décompositions possibles de L)

GN(L) :-

 L=[X1:[X2]],
 Article(X1),
 Nom(X2).

GV(L) :-

 L=[X1:L],
 Verbe(X1),
 GN(L).

Article(the).

Nom(man).

Nom(ball).

Verbe(hit).

Verbe(took).

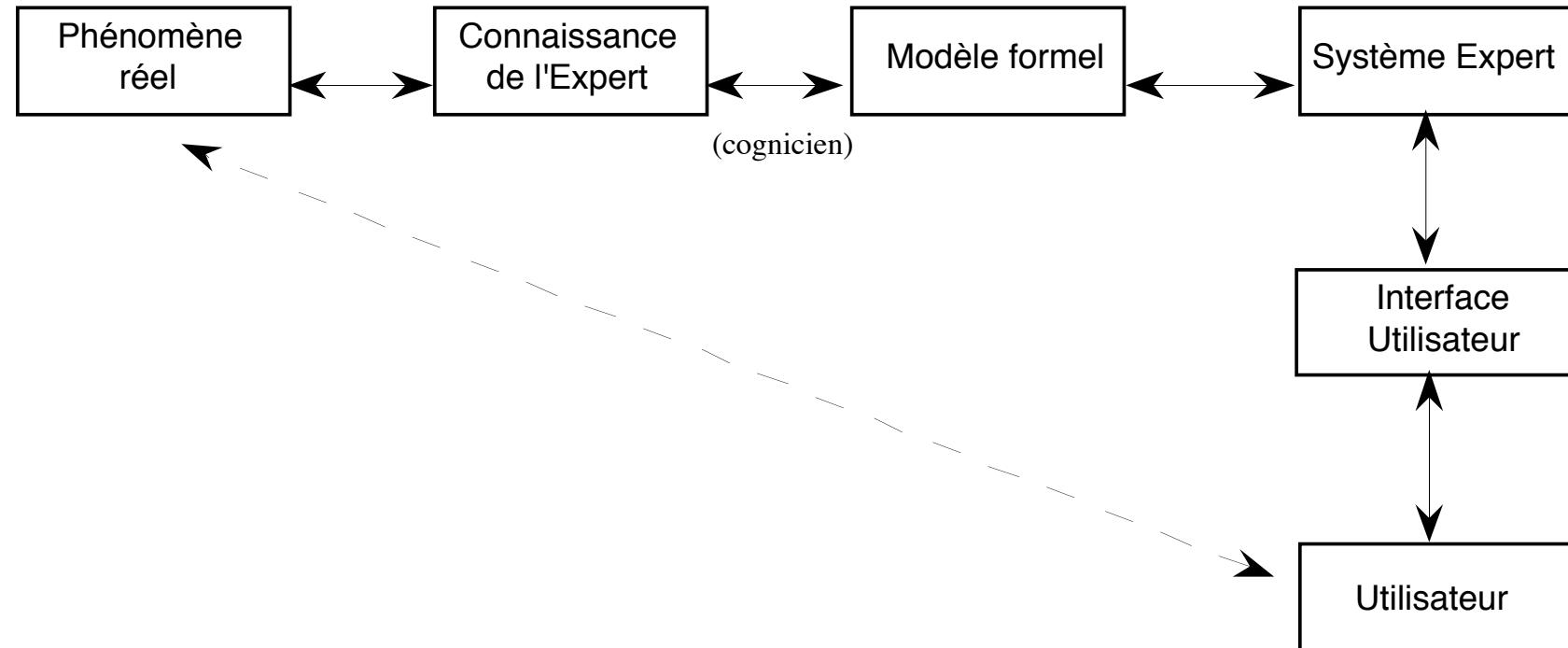
La caractère *additif* d'un langage objets tel que TurboProlog prend ici tout son intérêt : *il suffira d'ajouter de nouvelles clauses (sans modifier les précédentes) pour "apprendre" au programme de nouveaux mots ou de nouvelles règles de grammaire.*



Systèmes Experts



Systèmes Experts - Paradigme du système expert



Systèmes Experts - Champ d'application

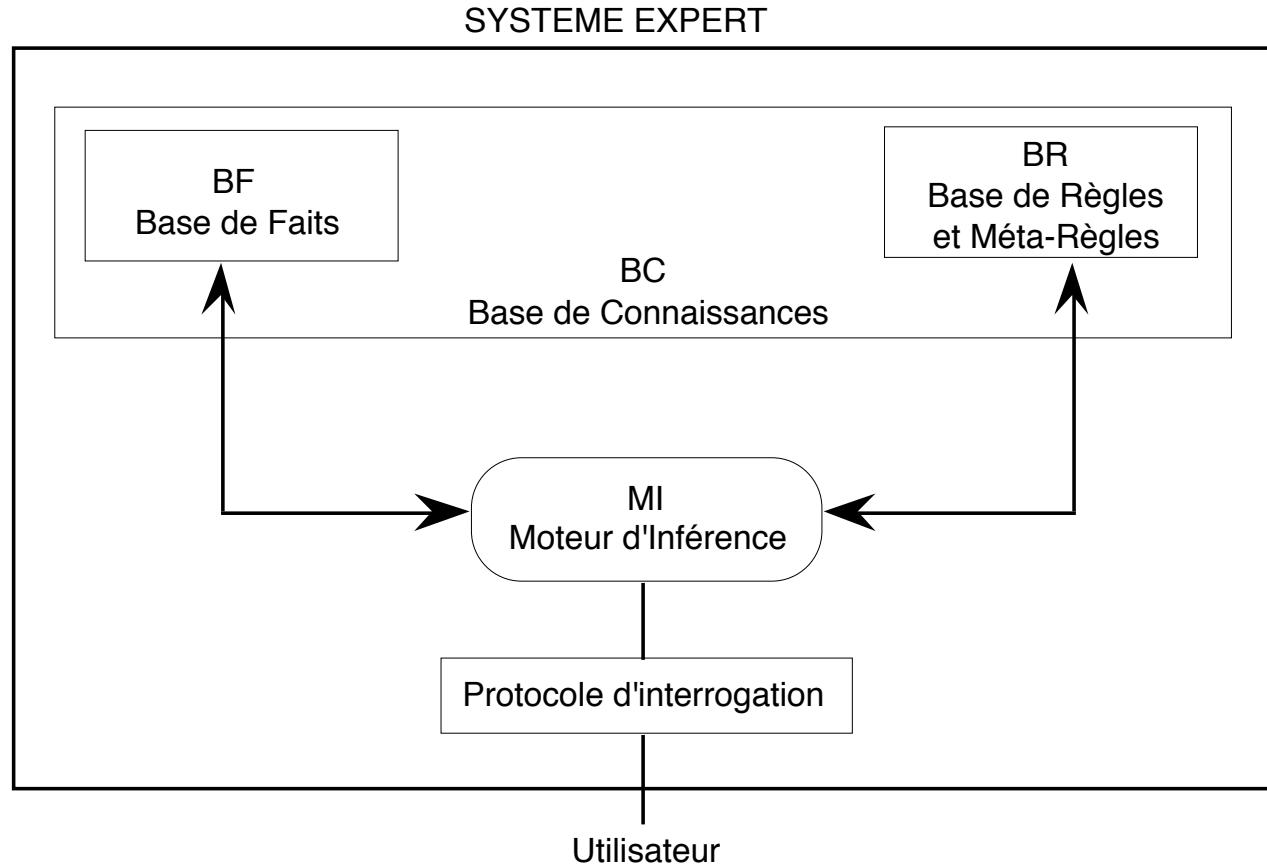
Les systèmes experts ont pour but de modéliser puis de simuler, dans un logiciel, le savoir - ou le "savoir faire" - d'un expert humain dans un domaine donné.

Leur champ d'application est vaste : il permettront par exemple d'aborder des problèmes pour lesquels :

- il n'existe pas de solution algorithmique connue, possible ou souhaitable,
- les connaissances mises en oeuvre sont de nature intuitive,
- les connaissances mises en oeuvre ne sont pas consignées explicitement par écrit,
- il existe de nettes différences de performance entre un individu moyen et un expert,
- les connaissances mises en oeuvre sont de nature qualitative plutôt que quantitative,
- les connaissances mises en oeuvre sont en évolution rapide et constante,
- la résolution implique des coûts élevés,
- la résolution se fait dans des conditions difficiles, voire stressantes,
- les données sont imprécises, voire incomplètes, ...



Systèmes Experts - Composantes du système expert



Systèmes Experts – Objectifs communes aux SE

Objectif 1 :

Capturer aisément les unités de savoir Faire

Objectif 2 :

Exploiter l'ensemble des unités de Savoir-faire :

- combiner des règles pour inférer des connaissances telles que : jugements, plans, preuves, décisions, prédictions, nouvelles règles, ...
- Rendre compte de la manière dont les nouvelles connaissances ont été inférées.

Objectif 3 :

Supporter aisément la révision de l'ensemble des unités de Savoir Faire.



Systèmes Experts - Les composantes

- La base de faits
- La base de règles
- Les métarègles et la métaconnaissance
- La représentation des connaissances incertaines



Systèmes Experts - La Base de connaissance

Initialement, la base de connaissance (BC) contient :

1- des faits avérés»

2- des faits à établir (expression de problèmes ou buts)

3- les connaissances opératoires associées au domaine

Ce sont des règles exprimées sous la forme :

Si : <condition 1>

 <condition 2> ...

 <condition n>

Alors : <conséquence 1>

 <conséquence 2> ...

 <conséquence p>

Conditions de déclenchement de la règle

Corps de la règle



Systèmes Experts - Exemple

-Soit la base de connaissance

-Base de règles :

R5 : si Z et L alors S

R1 : si A et N alors E

R3 : si D ou M alors Z

R2 : si A alors M

R4 : si Q et (non W) et (non Z) alors N

R6 : si L et M alors E

R7 : si B et C alors Q

-Base de faits :

A

L

Dessiner le Diagramme chainage avant depuis les faits avérés A et L, et de chainage arrière avec but à prouver E.



Systèmes Experts - Structures de contrôle

□ Cycle du moteur d'inférence :

- Phase de d'évaluation :
 - sélection
 - filtrage
 - résolution de conflits
- Phase d'exécution

□ Mode de raisonnements :

- Chaînage avant (données)
- Chaînage arrière (but)
- Chaînage mixte



Systèmes Experts – La phase d'évaluation

La phase de sélection :

Détermine à partir d'un état présent ou passé de la base de faits (BF), et d'un état présent ou passé de la base de règles (BR) :

- un sous-ensemble F1 de BF, et
- un sous-ensemble R1 de BR

qui méritent d'être comparés lors de l'étape de filtrage.

A ce stade, certaines stratégies peuvent être mises en œuvre pour sélectionner de préférence certains faits ou règles plutôt que d'autres (exemple : sélectionner dans F1 les faits déduits les plus récents).

Le filtrage (pattern matching) :

Le moteur d'inférence compare la partie déclencheur de chacune des règles de R1 par rapport à l'ensemble F1 des faits.

Un sous-ensemble R2 de R1 rassemble les règles jugées compatibles avec F1.

R2 est appelé **l'ensemble de conflit**.



Systèmes Experts - La Phase d'évaluation

La résolution de conflits :

Le moteur détermine l'ensemble R3 des règles (sous-ensemble de R2) qui peuvent être effectivement déclenchées.

Si R3 est vide, alors il n'y a pas de phase d'exécution pour ce cycle.

Au terme de ces différentes étapes de la **phase d'évaluation** (sélection, filtrage, résolution de conflits), est lancée la **phase d'exécution** qui met en œuvre les actions définies par les règles R3.



Systèmes Experts - Types de Systèmes Experts

Haton & Haton (1989) distinguent différents types de systèmes experts :

□ Systèmes d'interprétation de données.

- P.ex., systèmes de diagnostic en médecine ("de quelle maladie s'agit-il?"), système d'interprétation géologique ("les mesures seismologiques permettent-elles de croire à l'existence de dépôts minéraux importants?"), systèmes d'évaluation psychologique ("s'agit-il d'un cas suicidaire?"), etc.

□ Systèmes de prédiction.

- P.ex., systèmes de prédiction météorologique ("Il pleut aujourd'hui en France. Va-t-il pleuvoir en Suisse demain?"), prédictions géopolitiques ("Les conflits de guerre sont particulièrement fréquents en situation de crise économique. Quelles combinaisons précises de facteurs économiques, sociologiques et politiques prédisent un déclenchement d'hostilités?"), etc.



Systèmes Experts - Types de Systèmes Experts

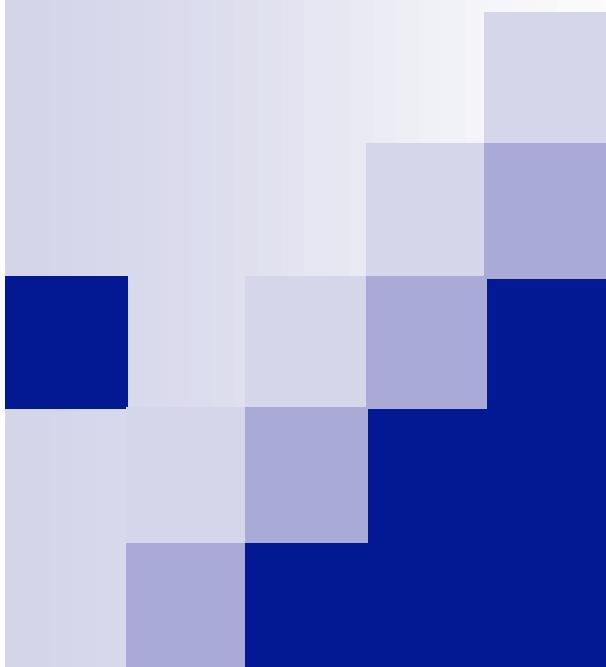
□ Systèmes de planification.

- P.ex., système de réservation de vols aériens, planification des altitudes de vol selon les vents connus et les corridors disponibles, planification des actions d'assemblage d'un robot industriel, planification des interventions requis pour la construction d'un bâtiment, etc.

□ Systèmes de conception.

- P.ex., Développement et simplification de circuits intégrés, aménagement d'une cuisine optimale dans un espace donné, clonage de gènes, création d'un nouveau composé chimique, etc.





Constraints Satisfaction Problems



Constraints Satisfaction Problems - Objectifs

La programmation par contrainte (CSP) est à l'interface de l'Intelligence Artificielle et de la Recherche Opérationnelle. Elle s'intéresse aux problèmes définis en termes de contraintes de temps, d'espace, ... ou plus généralement de ressources :

- **les problèmes de planification et ordonnancement** : planifier une production, gérer un trafic ferroviaire, ...
- **les problèmes d'affectation de ressources** : établir un emploi du temps, allouer de l'espace mémoire, du temps cpu par un système d'exploitation, affecter du personnel à des tâches, des entrepôts à des marchandises, ...
- **les problèmes d'optimisation**: optimiser des placements financiers, des découpes de bois, des routages de réseaux de télécommunication, ...

Ces problèmes ont la particularité commune d'être fortement combinatoires: il faut envisager un grand nombre de combinaisons avant de trouver une solution. La programmation par contraintes est un ensemble de méthodes et algorithmes qui tentent de résoudre ces problèmes de la façon la plus efficace possible.



Constraints Satisfaction Problems - Définition

Un réseau de contraintes à domaines finis $R = \{X, D, C\}$ est défini par :

– *un ensemble de variables :*

$$X = \{x_1, \dots, x_n\},$$

– *un ensemble de domaines finis :*

$D = \{D_1, \dots, D_n\}$ où D_i est l'ensemble des valeurs pouvant être affectées à la variable x_i ; on le note également $D(x_i)$,

– *un ensemble de contraintes :*

$C = \{C_1, \dots, C_m\}$. Chaque contrainte C_i est définie sur un sous-ensemble de variables $\text{var}(C_i) = (x_{i1}, \dots, x_{ik})$ de X par un sous-ensemble $\text{rel}(C_i)$ du produit cartésien $D_{i1} \times \dots \times D_{ik}$ qui spécifie les combinaisons autorisées de valeurs pour les variables $\{x_{i1}, \dots, x_{ik}\}$.



Constraints Satisfaction Problems - Vocabulaire

- Chaque variable x_i a un domaine non vide de valeurs possibles.
- Un Etat est une affectation de valeurs pour quelques unes ou toutes les variables.
- **Une affectation est consistante** s'il y a aucune violation de contrainte.
- **Une affectation est complète** si toutes les variables ont une valeur.
- **Une Solution** est une affectation Complète et Consistante.



Constraints Satisfaction Problems - Variables, contraintes

Types de variables :

- **Variables discrètes**
 - Domaines finis : nombre de variables n , booléens.
 - Domaines infinis : entiers, chaînes de caractères, etc...

Ex: Ordonnancement de tâches; les variables sont les jours de début et de fin des tâches.
- **Variables continues**

Ex: temps (mesure précise du temps de début et de fin d'une tâche), données mesurables, ...

Types de contraintes : Une contrainte est caractérisée par son *arité*, c'est-à-dire le nombre de variables qu'elle implique,

- **Unaire**: Les contraintes ne concernent qu'une variable.
- **Binaire**: Les contraintes concernent deux variables.
- **Multiple** : Les contraintes concernent 3 variables ou plus.
- **Contraintes de préférence** : ex. , pour la colorisation d'une carte, contrainte du type « préférer le rouge au bleu »...



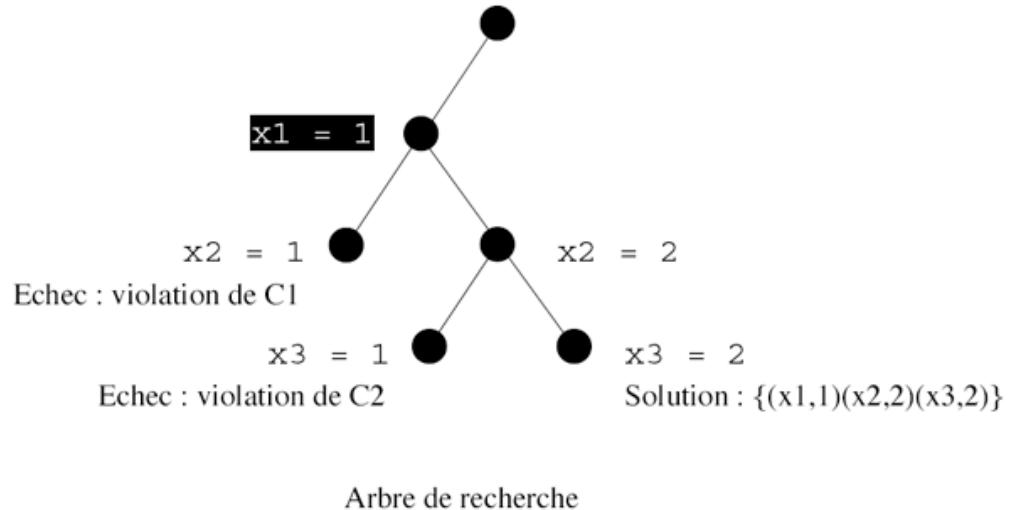
Constraints Satisfaction Problems - Backtrack

L'algorithme de recherche de solution le plus simple est celui du **Backtrack** : les variables du problème sont instanciées avec les valeurs des domaines, dans un ordre prédéfini, jusqu'à ce que l'un de ces choix ne satisfasse pas une contrainte. Dans ce cas, on doit remettre en cause la dernière instantiation réalisée. Une nouvelle valeur est essayée pour la dernière variable instanciée, que l'on appelle la variable courante.

Si toutes les valeurs du domaine de cette variable ont été testées sans succès, on doit procéder à un backtrack : on choisit une autre valeur pour la variable précédant immédiatement la variable courante. On répète ce processus jusqu'à obtenir une solution, c'est à dire une instantiation de toutes les variables. Si on a parcouru tout l'arbre de recherches sans la trouver, alors on a prouvé que le problème n'a pas de solution.

Exemple :

- soient x_1 , x_2 et x_3 trois variables,
- soient $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$ leurs domaines respectifs.
- On pose les contraintes suivantes:
 $C_1 = [x_1 < x_2]$ et $C_2 = [x_2 = x_3]$.
- On suppose que l'on instancie les variables dans l'ordre croissant des indices, en choisissant la plus petite valeur d'abord.
 $x_1 = 1$



Constraints Satisfaction Problems - Filtrage

Dans le but de **réduire la taille de l'arbre de recherche**, une méthode de résolution spécifique est associée à chaque contrainte : son **algorithme de filtrage**. Cet algorithme vise à supprimer les valeurs des domaines des variables impliquées dans cette contrainte qui, compte tenu des autres domaines, ne peuvent appartenir à une solution du sous-problème défini par la contrainte. Ces suppressions évitent de parcourir des branches de l'arbre qui ne peuvent aboutir à une solution.

En d'autres termes elle permettent de **réduire l'espace de recherche**. Effectuer un filtrage efficace est une condition nécessaire à la résolution d'applications.

Exemple :

- soient x_1 et x_2 deux variables,
- soient $D(x_1) = D(x_2) = \{1, 2, 3\}$ leurs domaines respectifs,
- soit la contrainte $C = [x_1 < x_2]$.

La valeur 3 peut être supprimée de $D(x_1)$, car il n'existe aucune valeur du domaine de x_2 telle que C soit satisfaite si on instancie x_1 avec 3 (et idem pour la valeur x_2 : la valeur 1 peut être supprimée).



Constraints Satisfaction Problems - Propagation

Après chaque modification du domaine d'une variable il est nécessaire d'étudier à nouveau l'ensemble des contraintes impliquant cette variable, car la modification peut conduire à de nouvelles déductions.

C'est ce que l'on appelle la propagation.

Exemple :

- soient x_1 , x_2 et x_3 trois variables,
- soient $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$ leurs domaines respectifs,
- soient les contraintes $C_1 = [x_1 < x_2]$ et $C_2 = [x_2 = x_3]$.

Le filtrage de la valeur 1 de $D(x_2)$ relatif à C_1 peut être propagée sur $D(x_3)$: si la valeur 1 n'appartient plus à $D(x_2)$, alors la valeur 1 peut également être supprimée de $D(x_3)$, car il n'existe plus de solution de C_2 telle que x_3 soit instanciée avec 1.



Constraints Satisfaction Problems - Heuristiques

Les méthodes heuristiques permettent de simplifier les recherches:

- **Choix des variables:**

- **Heuristique du nombre de valeurs restantes minimum**
(minimum remaining values (MRV)).
Choisir la variable avec le moins de valeurs possibles.

- **Heuristique du degré**
Choisir la variable présente dans le plus de contraintes.

- **Choix des valeurs:**

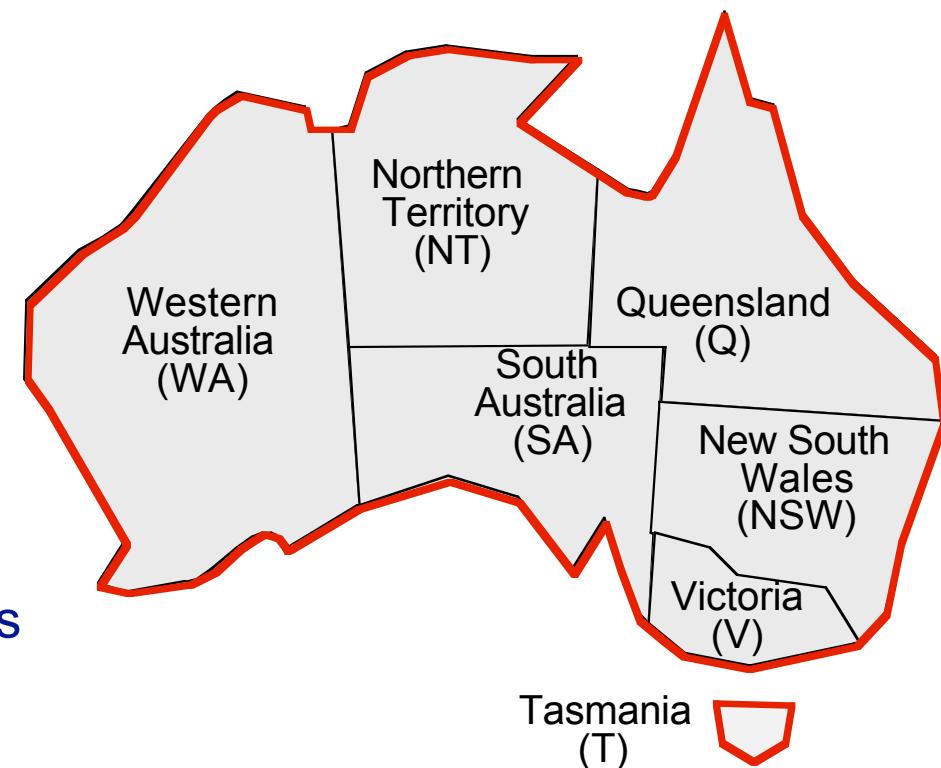
- **Heuristiques des valeurs les moins contraignantes**
(least-constraining value)
Choisir la valeur qui va enlever le moins de choix pour les variables voisines.



Constraints Satisfaction Problems - Exemple

Coloration de carte :

- **Variables** : WA, NT, SA, Q, NSW, V, T
- **Domaines** : $D_i = \{\text{rouge}, \text{vert}, \text{bleu}\}$
- **Contraintes** : Les régions adjacentes doivent avoir des couleurs différentes



D'après S.Paquet 2003

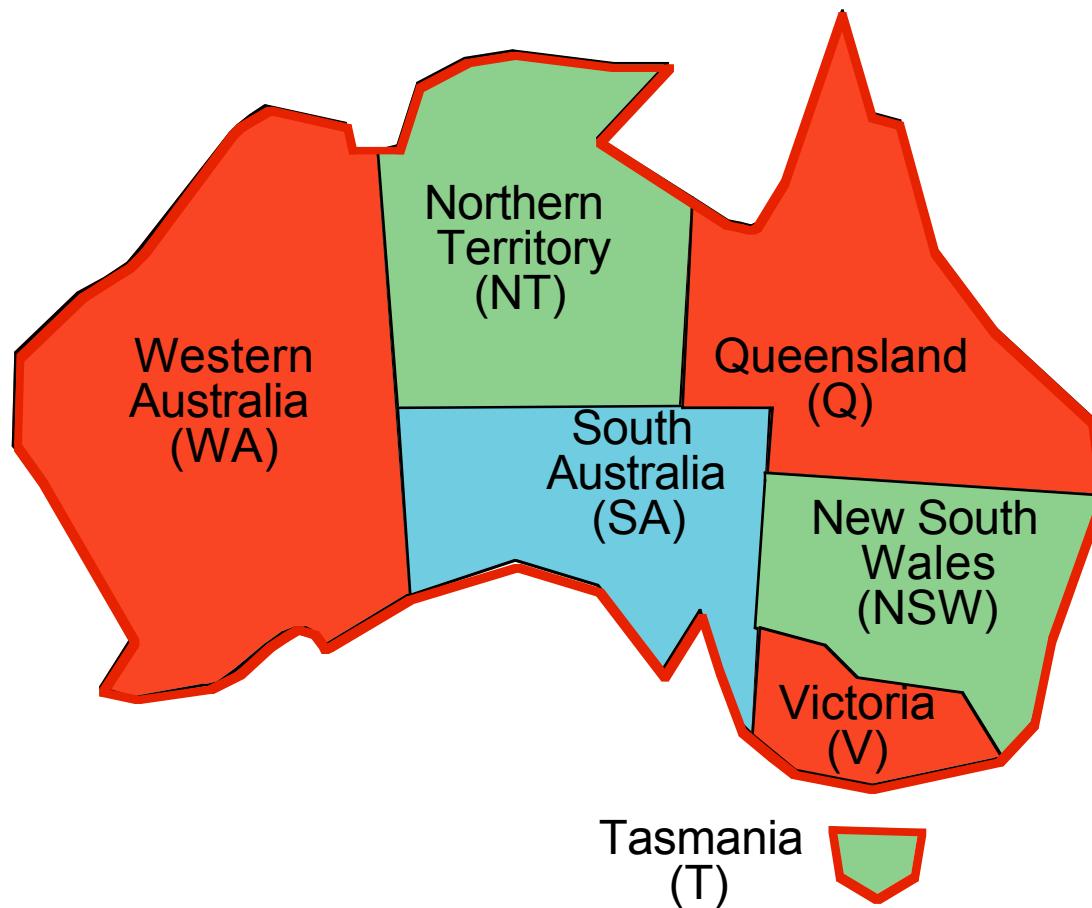
Olivier Boisard - Cours d'Intelligence Artificielle



Constraints Satisfaction Problems - Coloration de carte

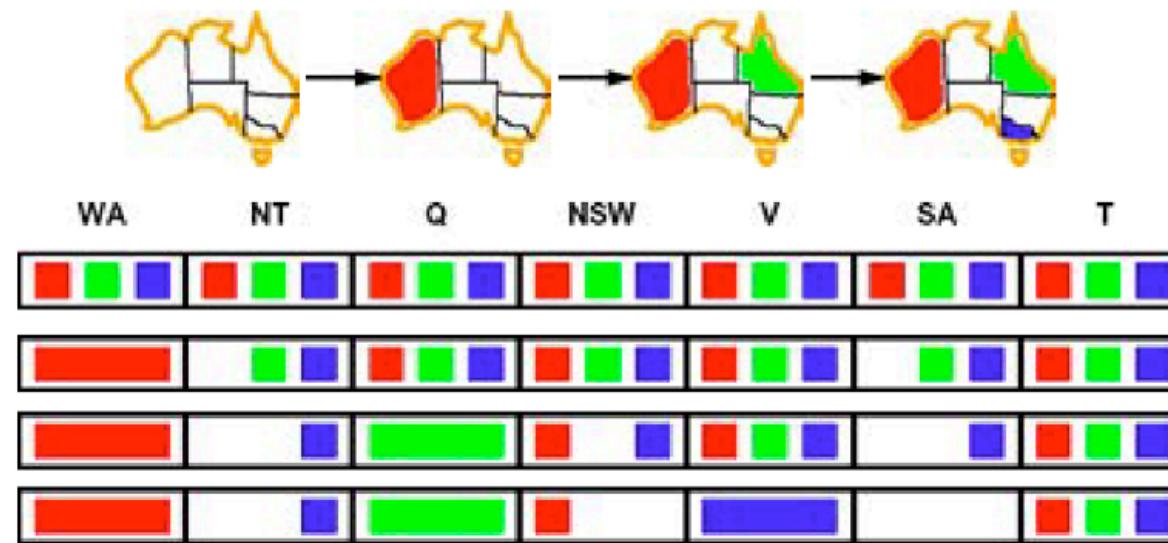
Exemple de solution
Satisfaisant à toutes
Les contraintes :

{WA=rouge, NT=vert,
Q=rouge, SA=bleu,
NSW=vert, V=rouge,
T=vert}



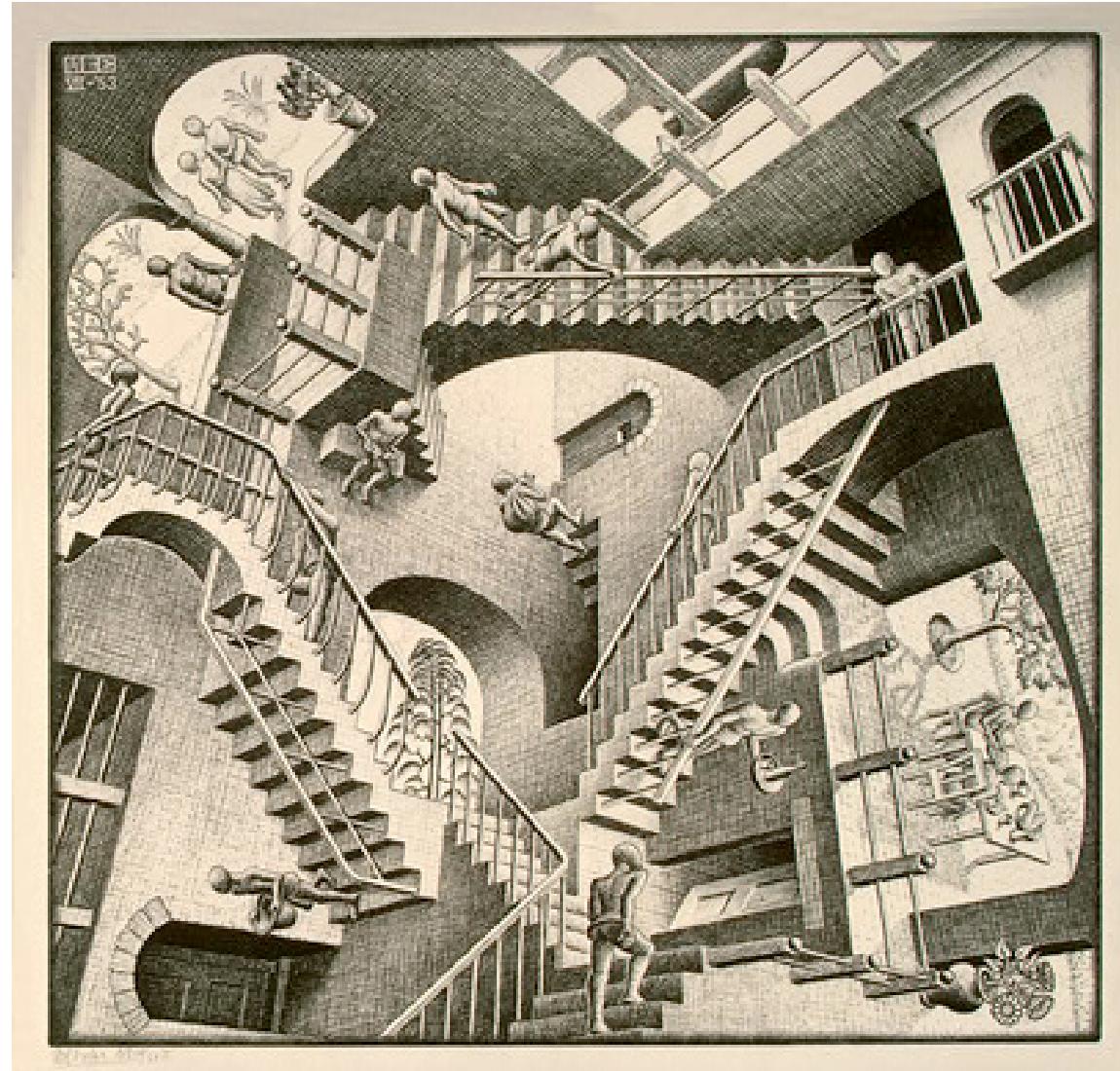
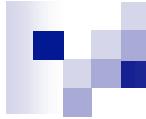
Constraints Satisfaction Problems - Forward Checking

Le **forward checking** est une stratégie "avant", qui avant d'affecter une valeur v à une variable, vérifie que v est compatible avec les variables *suivantes*, c'est-à-dire qu'il existe au moins une valeur pour chaque variable suivante qui soit consistante avec v (contrairement au *retour arrière*, qui vérifie que la valeur de la variable courante est compatible avec les valeurs affectées aux variables *précédentes*).



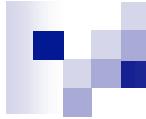
Conclusion ...





Exemples ...





Réaliser un programme prolog modélisant un grammaire formelle permettant de produire des phrases avec propositions relatives, telles que :

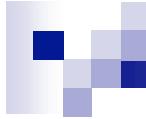
« l'homme qui a vu l'homme »

« l'homme qui a vu l'homme qui a vu l'homme »

« l'homme qui a vu l'homme qui a vu l'homme qui a vu l'homme »,
etc ...

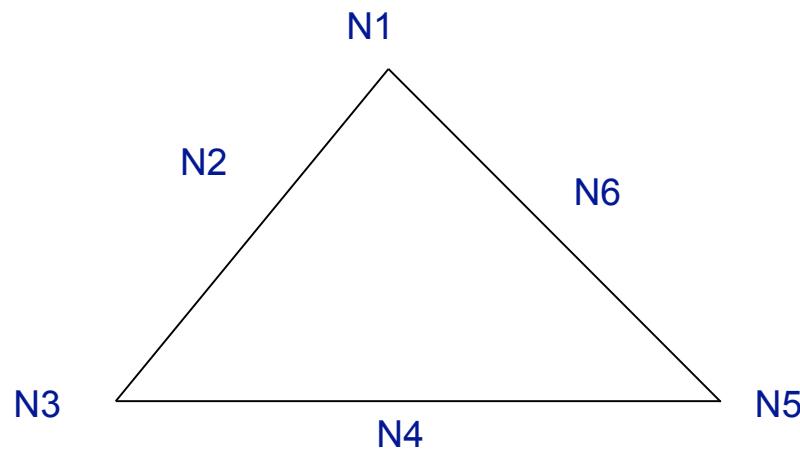
- Définir les éléments du système formel
- Formaliser l'arbre de dérivation
- Ecrire le programme en prolog





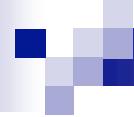
Nombres en triangle

On dispose les nombres 1 à 6 en triangle, en n'utilisant qu'une fois chaque nombre :



Trouver toutes les manières de placer les nombres sur le triangle de telle sorte que la somme des trois nombres figurant sur un côté soit la même pour les trois cotés du triangle.





Nombres en triangle

La clause « choisir » :

choisir (X,[X:Xs], Xs).

choisir (X,[Y,Xs],[Y:Zs]) :- choisir (X,Xs,Zs)

La relation « choisir (X,Xs,Ys) consiste à choisir un élément X dans Xs, Ys est alors la liste obtenue à partir de Xs quand on ôte l'élément X.

La première clause correspond au cas où l'élément choisi est le premier de la liste, la deuxième au cas où l'élément choisi est dans la queue de la liste.



Nombres en triangle

choisir (X,[X:Xs], Xs).

choisir (X,[Y,Xs],[Y:Zs]) :- choisir (X,Xs,Zs)

solution([N1,N2,N3,N4,N5,N6]) :-

 choisir(N1,[1,2,3,4,5,6],D1),

 choisir(N2,D1,D2),

 choisir(N3,D2,D3),

 S is N1+N2+N3,

 choisir(N4,D3,D4),

 N5 is S – N3 – N4,

 choisir(N5,D4,D5),

 N6 is S-N1-N5,

 choisir(N6,D5,_).



Exemple de système expert pour la résolution d'un problème d'aménagement

-Le système comprend :

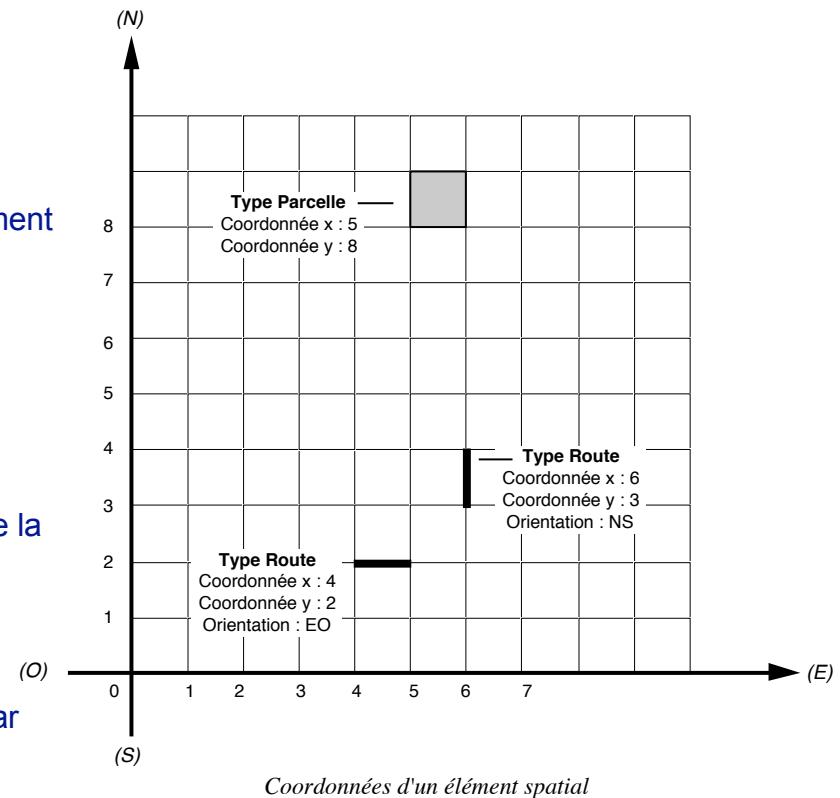
- une *structure générique* décomposant tout élément spatial en (cf. diagramme) :

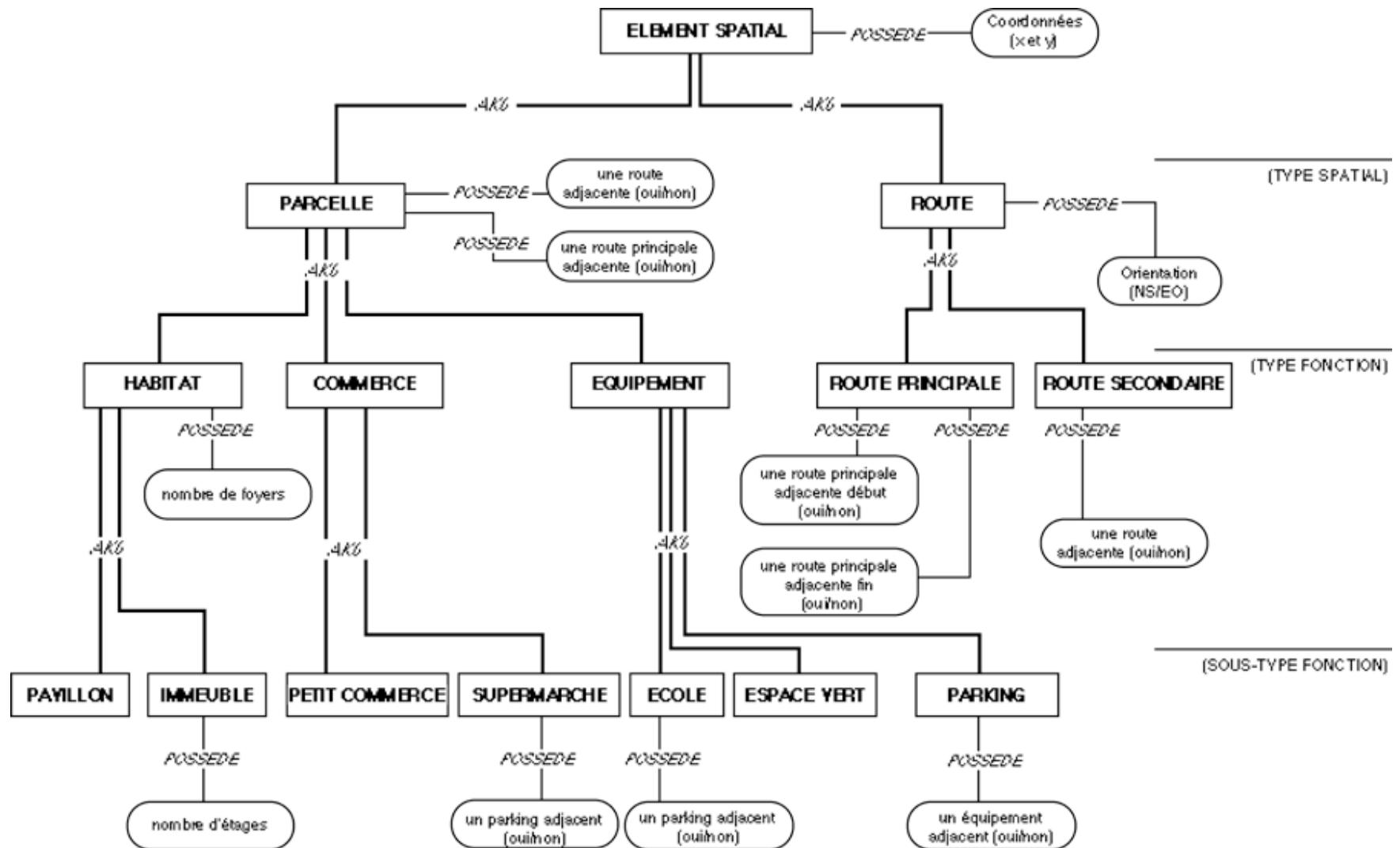
- Type Spatial (Parcelle ou Route),
- Type Fonctionnel (Habitat, Commerce, Équipement, etc ...)
- Sous-Type Fonctionnel (Pavillon, Immeuble, École, Espace vert, etc ...)

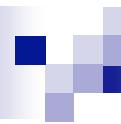
- des *objets instanciés* calquées sur le modèle de la *structure générique*.

- Les lettres A,B,C,D, ... représentent des variables *objets instanciés*.

- Toute valeur relative à un objet instancié A est représentée par son nom dans la structure générique suivie de "(A)".







Exemple de notation :

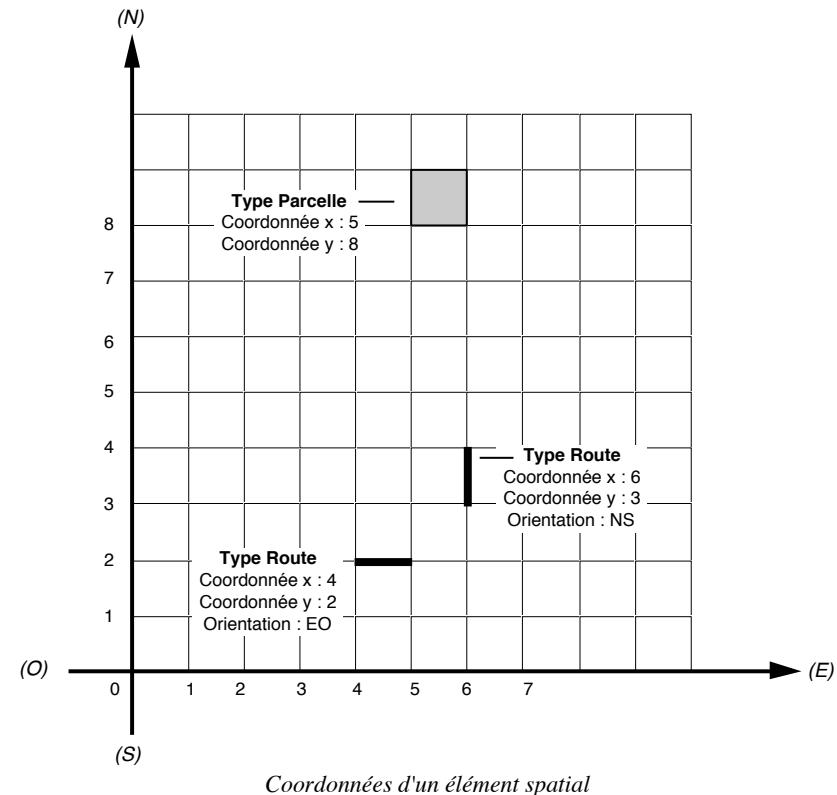
L'objet A="Immeuble de dix étages localisé en (3,4)" sera décrit dans la base de faits par:

ELEMENT_SPATIAL-COORDONNEE_X (A)= 3
ELEMENT_SPATIAL-COORDONNEE_Y (A)= 4
TYPE_SPATIAL (A) = PARCELLE
TYPE_FONCTIONNEL (A) = HABITAT
SOUS_TYPE_FONCTIONNEL (A) = IMMEUBLE
IMMEUBLE-NOMBRE_ETAGES (A) = 10

Remarque :

Si la seule information "IMMEUBLE-NOMBRE_ETAGES (A) = 10" était rentrée dans la base de fait, un "démon" serait activé pour en déduire :

SOUS_TYPE_FONCTIONNEL (A) = IMMEUBLE
TYPE_FONCTIONNEL (A) = HABITAT
TYPE_SPATIAL (A) = PARCELLE





Toute Règle du système expert est de la forme :

RÈGLE NUMÉRO (numéro de la règle)

SI (condition 1),
 (condition 2),
 (condition 3),

...

ALORS

 (conséquence 1),
 (conséquence 2),
 (conséquence 3),

...

FIN DE RÈGLE

La règle ne sera activée que si *toutes* les *conditions* sont remplies (ces conditions sont donc reliées implicitement par l'opérateur logique "ET"). *Toutes* les conséquences seront alors réalisées.



Les conséquences affectent de nouvelles valeurs aux variables (ou créent ces variables si elles ne sont pas encore définies pour l'objet considéré), avec l'opérateur "<=" .

Exemples :

TYPE_FONCTIONNEL (A) <= HABITAT (l'objet A a pour type fonctionnel HABITAT)

IMMEUBLE-NOMBRE_ETAGES (A) <= 6 (l'objet A est un immeuble de six étages)

Le système admettra également deux autres conséquences particulières :

CRÉER_FAIT (A) : crée (comme son nom l'indique), un nouveau fait A dans la base de faits,

FIN RECHERCHE : stoppe le processus de recherche (but atteint)





EXEMPLE DE BASE DE CONNAISSANCE :

BASE DE FAITS :

Il existe déjà un immeuble de dix étages en coordonnées (3,4) :

IMMEUBLE_NOMBRE_ETAGES (Élément1) = 10

ELEMENT_SPATIAL-COORDONNEE_X (Élément1)= 3

ELEMENT_SPATIAL-COORDONNEE_Y (Élément1)= 4

Il existe déjà une école en coordonnées (20,16). :

SOUS-TYPE_FONCTION (Élément2) = ÉCOLE

ELEMENT_SPATIAL-COORDONNEE_X (Élément2)= 20

ELEMENT_SPATIAL-COORDONNEE_Y (Élément2)= 16





EXEMPLE DE BASE DE CONNAISSANCE :

BASE DE REGLES :

Les règles expriment les principes à respecter dans l'aménagement des éléments spatiaux:

- principe (1) : toute parcelle doit être à proximité d'une route (principale ou secondaire), la créer si elle n'existe pas.
- principe (2) : toute habitation doit être à moins de 3 Km d'un commerce, le créer s'il n'existe pas.
- principe (3) : tout supermarché doit être à proximité d'un parking. Le créer s'il n'existe pas et qu'il existe une parcelle libre à proximité. S'il n'existe pas de parcelle libre, changer la localisation du supermarché.
- principe (4) : tout supermarché doit avoir au moins une route principale adjacente.
- principe (5) : toute route doit avoir une route principale ou secondaire adjacente.
- principe (6) : toute route principale doit avoir une route principale adjacente en début et en fin (les routes principales sont toujours continues),
- principe (7) : toute habitation doit être à moins de deux Km d'une école.
- etc ...
- principe (n) : la recherche est terminée lorsque tous les principes précédents sont respectés, et que toutes les parcelles sont occupées.



Exemple de traduction (partielle) du principe (1) :

Cette règle permet, si la parcelle n'a pas de route adjacente, d'en créer une par défaut sur sa frontière sud.

RÈGLE NUMÉRO 1 :

SI PARCELLE_ROUTE_ADJACENTE (A) = NON
ALORS CREER_FAIT (B),
 TYPE_SPATIAL(B) = ROUTE
 ELEMENT_SPATIAL-COORDONNEE_X(B)<=ELEMENT_SPATIAL-COORDONNEE_V(A)
 ELEMENT_SPATIAL-COORDONNEE_Y(B)<=ELEMENT_SPATIAL-COORDONNEE_Y (A)
 ROUTE_ORIENTATION (B) <= EO
FIN DE RÈGLE



Exemple de traduction (partielle) du principe (3) :

Cette règle permet, de créer un parking au nord d'un supermarché si aucun parking n'est déjà à proximité du supermarché .

RÈGLE NUMÉRO 2 :

SI

SOUS-TYPE_FONCTIONNEL (A) = SUPERMARCHÉ

(traduction : soit A un supermarché)

SOUS-TYPE_FONCTIONNEL (B) = PARKING

ELEMENT_SPATIAL-COORDONNEE_X(B) = CONNU

ELEMENT_SPATIAL-COORDONNEE_Y(B) = CONNU

(traduction : soit un parking B dont les coordonnées sont connues)

ELEMENT_SPATIAL-COORDONNEE_X(B) <> ELEMENT_SPATIAL-COORDONNEE_X(A)

ELEMENT_SPATIAL-COORDONNEE_Y(B) <> ELEMENT_SPATIAL-COORDONNEE_Y(A) +1

(traduction : le parking B n'est pas au nord du supermarché)

ELEMENT_SPATIAL-COORDONNEE_X(B) <> ELEMENT_SPATIAL-COORDONNEE_X(A)

ELEMENT_SPATIAL-COORDONNEE_Y(B) <> ELEMENT_SPATIAL-COORDONNEE_Y(A) +1

(traduction : le parking B n'est pas à l'est du supermarché)

ELEMENT_SPATIAL-COORDONNEE_X(B) <> ELEMENT_SPATIAL-COORDONNEE_X(A) -1

ELEMENT_SPATIAL-COORDONNEE_Y(B) <> ELEMENT_SPATIAL-COORDONNEE_Y(A)

(traduction : le parking B n'est pas au sud du supermarché)

ELEMENT_SPATIAL-COORDONNEE_X(B) <> ELEMENT_SPATIAL-COORDONNEE_X(A)

ELEMENT_SPATIAL-COORDONNEE_Y(B) <> ELEMENT_SPATIAL-COORDONNEE_Y(A) -1

(traduction : le parking B n'est pas à l'ouest du supermarché)

NON (EXISTE_PARCELLE (ELEMENT_SPATIAL-COORDONNEE_X(A) ,

ELEMENT_SPATIAL-COORDONNEE_Y(A))+1

(traduction : la parcelle au nord du supermarché est libre)

ALORS CREER FAIT (C)

SOUS-TYPE_FONCTION (C) <= PARKING

ELEMENT_SPATIAL-COORDONNEE_X(C) <= ELEMENT_SPATIAL-COORDONNEE_X(A)

ELEMENT_SPATIAL-COORDONNEE_Y(C) <= ELEMENT_SPATIAL-COORDONNEE_Y(A)+1

FIN DE RÈGLE



- Concaténer deux listes :

```
concat([],L2,L3) :- L2=L3.  
concat(L1,L2,L3) if  
    L1=[X:L4],  
    concat(L4,L2,L5),  
    L3=[X:L5].
```

- Insérer un élément dans une liste :

Clause inserer (X,L2,L3), où X est inséré dans la liste triée en ordre croissant L2 pour obtenir L3.

```
inserer(X,[],[X]).  
inserer(X,[X:Ys],[Y:Zs]) :- X > Y,  
inserer(X,Ys,Zs).Inserer(X,[Y:Ys],[X,[Y:Ys]]) :-  
X=<Y.
```



Concaténer deux listes :

```
appart(X,L) :-  
            L=[X:_].          (a)  
appart(X,L) :-  
            L=[_|L1],  
            appart(X,L1)    (b)
```



Réunir deux listes :

```
concat([],L2,L3) :-  
    L2=L3.
```

```
concat(L1,L2,L3) :-  
    L1=[X:L4],  
    appart(X,L2],  
    concat(L4,L2,L3).
```

```
concat(L1,L2,L3) :-  
    L1=[X:L4],  
    concat(L4,L2,L5),  
    L3=[X:L5].
```

(condition initiale)

(ignorer les doublons)

(= soit X la tête de L1)

(= et X déjà dans L2)

(= alors poursuivre en ignorant X)



Renverser une liste :

```
renverser(L1,L2) :-  
    L1=[],  
    L2=[].
```

```
renverser(L1,L2) :-  
    L1=[X:L3],  
    renverser(L3,L4),  
    concat(L4,[X],L2).
```

(condition initiale)
(= la liste vide renversée
est la liste vide ...)
(récursion)



Problème du singe et de la banane ...

Un singe se trouve à la porte d'une pièce. Il souhaite attraper une banane pendue au plafond au centre de la pièce. Il ne peut attraper la banane directement lorsqu'il est au sol, et ne peut le faire qu'en montant sur une boîte. Il peut aller chercher cette boîte ailleurs dans la pièce.

Comment modéliser ce problème ?



Problème du singe et de la banane ...

Les éléments du système :

- position du singe dans la pièce,
- position de la boîte dans la pièce,
- le singe est-il au sol ou sur la boîte ?
- le singe a-t-il ou non la banane ?

Formalisation des états du système :

Etat (X_1, X_2, X_3, X_4)

X_1 : position du singe dans la pièce (à la porte, à la fenêtre, etc ...)

X_2 : le singe est-il au sol ou sur la boîte ? (sur-le-sol, sur-la-boîte)

X_3 : position de la boîte dans la pièce (à la porte, à la fenêtre, etc ...).

X_4 : le singe a-t-il ou non la banane (possède, ne-possède-pas)

L'objectif est : `etat(_,_,_,possede)`

Les actions possibles :

- Agripper la banane
- Monter sur la boîte
- Pousser la boîte
- se déplacer



Problème du singe et de la banane ...

La relation prolog `deplacement` exprime toutes les modifications possibles de l'état du système :

`deplacement(Etat1, M, Etat2)`

`Etat1` : état avant le déplacement

`M` : mouvement,

`Etat2` : état après le déplacement



Problème du singe et de la banane ...

Les mouvements autorisés :

```
deplacement(  
etat(aucentre,surlaboite,aucentre,napas),  
agripper,  
etat(aucentre,surlaboite,aucentre,possede)).
```

```
deplacement(etat(P,surlesol,P,H),  
grimper,  
etat(P,surlaboite,P,H)).
```

```
deplacement(etat(P1,surlesol,P1,H),  
pousser(P1,P2),  
etat(P2,surlesol,P2,H)).
```

```
deplacement(etat(P1,surlesol,B,H),  
marcher(P1,P2),  
etat(P2,surlesol,B,H)).
```



Problème du singe et de la banane ...

Principe du programme du problème du singe et la banane :

```
deplacement(  
etat(aucentre,surlaboite,aucentre,napas),  
agripper,  
etat(aucentre,surlaboite,aucentre,possede)).
```

```
deplacement(etat(P,surlesol,P,H),  
grimper,  
etat(P,surlaboite,P,H)).
```

```
deplacement(etat(P1,surlesol,P1,H),  
pousser(P1,P2),  
etat(P2,surlesol,P2,H)).
```

```
deplacement(etat(P1,surlesol,B,H),  
marcher(P1,P2),  
etat(P2,surlesol,B,H)).
```

```
peutprendre(etat(_,_,_,possede)).
```

(le singe a la banane)

```
peutprendre(Etat1) :-
```

(il faut agir)

```
peplacement(Etat1,Déplacement, Etat2),
```

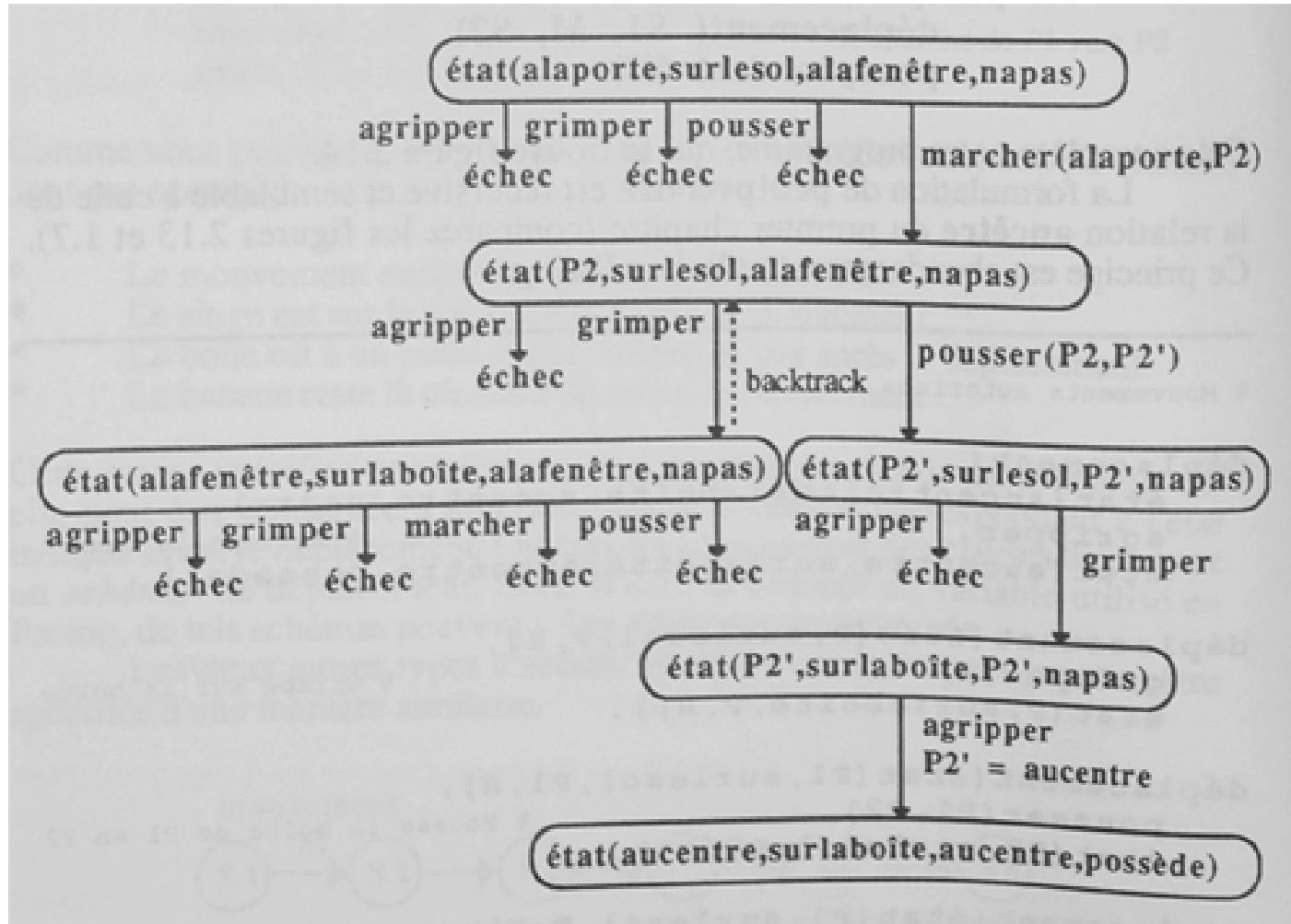
(action)

```
peutprendre(Etat2).
```

(on prend).



Problème du singe et de la banane ...



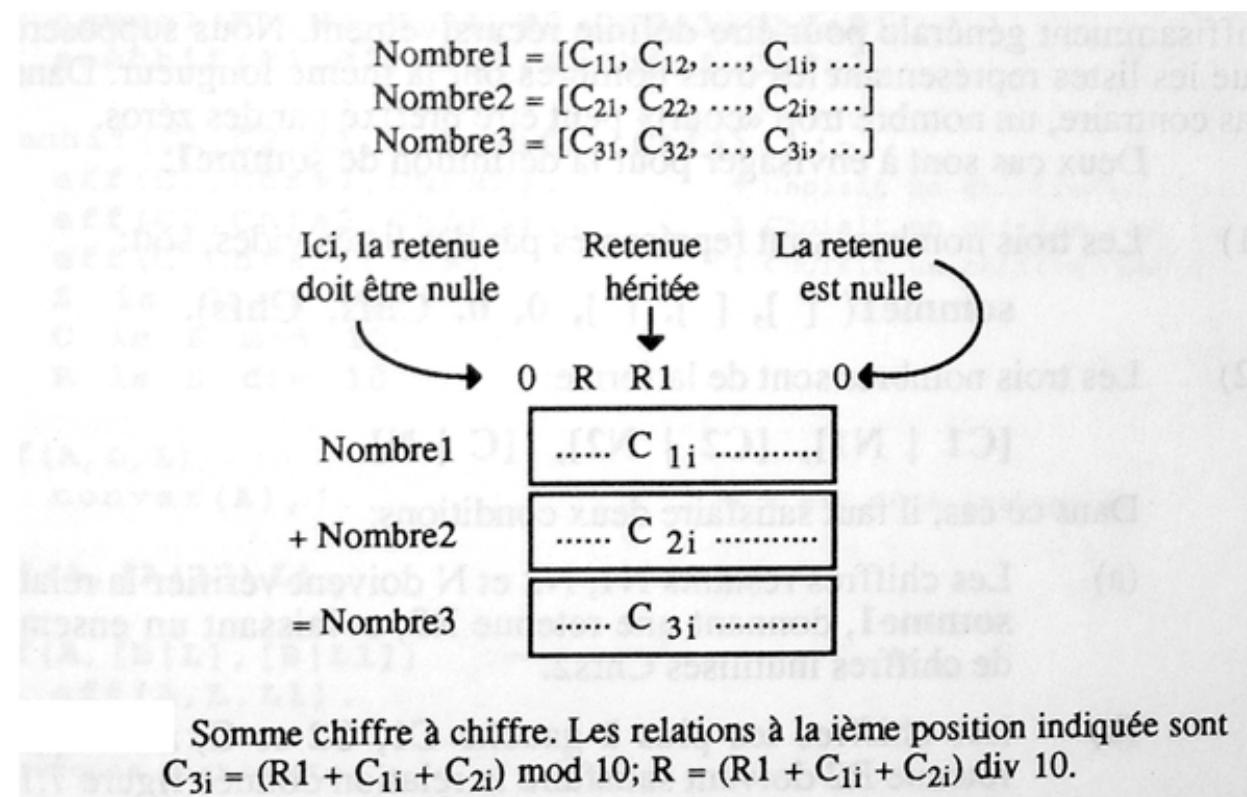
$$\begin{array}{r} \text{DONALD} \\ + \text{GERARD} \\ \hline = \text{ROBERT} \end{array}$$

Question : comment affecter un chiffre à chaque lettre D,O,N, etc .. de telle sorte que l'addition soit valide ?
(des lettres différentes auront des chiffres différents).



Créer une relation somme($N1, N2, N$), où $N1, N2, N$ représentent les trois nombres, avec $N=N1+N2$.

Somme([D,O,N,A,L,D],[G,E,R,A,R,D],[R,O,B,E,R,T]).





Création de la relation somme1 :

Somme1(N1,N2,N,R1,R,Chiffre1, Chiffres)

N1,N2 et N sont les nombres provenant de somme.

R1 la retenue provenant de la droite (avant addition de N1 et N2),

R la retenue propagée après addition.

Chiffre1 la liste des chiffres disponibles pour instancier les variables de
N1,N2 et N

Chiffres est la liste des chiffres non utilisés.

Exemple :

```
?- somme1([H,E],[6,E],[U,S],1,1,[1,3,4,7,8,9],Chiffres).
```

H=8

E=3

S=7

U=4

Chiffres=[1,9]

yes



Deux cas sont à envisager pour la définition de somme1 :

- (1) Les trois nombres sont représentés par des listes vides, soit :

Somme1([],[],[],0,0,Chfs,Chfs)

- (2) Les trois nombres sont de la forme :

[C1:N1],[C2:N2],[C:N]

Dans ce cas, il faut satisfaire deux conditions :

- (a) Le chiffres restants N1,N2, et N doivent vérifier la relation somme1, donnant une retenue R2, et laissant un ensemble de chiffres inutilisés Chfs2.
- (b) Les chiffres les plus à gauche C1,C2 et C, ainsi que la retenue R2 doivent satisfaire la relation donnée précédemment (slide prec.) : C est le résultat de l'addition de R2 , C1 et C2, qui produit aussi une retenue. La relation somchif aura en charge de vérifier cette condition.

Soit en prolog :

```
Somme1([C1:N1],[C2:N2],[D:N], R1,R,Chfs1,Chfs):-  
    somme1(N1,N2,N,R1,R2,Chfs1,Chfs2),  
    somchif(C1,C2,R2,C,R,Chfs2,Chfs).
```



```

% Résolution de casse-tête cryptarithmétiques

somme(N1,N2,N)      :-                      % Nombres sous forme de listes
    somme1(N1,N2,N,          % de chiffres
            0,0,                % Retenues: héritée, propagée
            [0,1,2,3,4,5,6,7,8,9],_). % Chiffres disponibles

somme1([],[],[],0,0,Chiffres,Chiffres).

somme1([C1|N1],[C2|N2],[C|N],R1,R,Chfs1,Chfs)  :-
    somme1(N1,N2,N,R1,R2,Chfs1,Chfs2),
    somchif(C1,C2,R2,C,R,Chfs2,Chfs).

somchif(C1,C2,R1,C,R,Chfs1,Chfs)  :-
    eff(C1,Chfs1,Chfs2),           % Choisit un chiffre pour C1
    eff(C2,Chfs2,Chfs3),           % Choisit un chiffre pour C2
    eff(C,Chfs3,Chfs),             % Choisit un chiffre pour C
    S is C1+C2+R1,
    C is S mod 10,
    R is S div 10.

eff(A,L,L)  :-                      % A est déjà instanciée
    nonvar(A), !.

eff(A,[A|L],L).

eff(A,[B|L],[B|L1])  :-              % A + B + L1
    eff(A,L,L1).

```



Tri de listes – première solution

Créer la relation Tri(Liste, Liste Triée)

- Trouver dans Liste deux éléments adjacents X et Y, tels que $X > Y$, échanger X et Y, pour donner Liste1, puis trier Liste1.
- S'il n'y a dans Liste aucun couple d'éléments X,Y adjacents tels que $X > Y$, alors Liste est trié.

tribulle(Liste,Triée) :-

 echange(Liste,Liste1),
 tribulle(Liste1,Triée).

(échange util dans la liste ?)

tribulle(triée,triée).

(non, alors elle est triée)

echange([X,Y:Reste],[Y,X:Reste]):-
 X>Y.

(échange 2 premiers éléments)

echange([Z:Reste],[Z:R1]):-
 echange(Reste,R1]).

(échange les éléments dans queue).



Tri de listes – seconde solution

Tri par insertion :

Pour trier $L=[X:Q]$

- (1) : il faut trier Q , la queue de L ,
- (2) : il faut insérer X dans Q triée, à un emplacement tel que la liste résultante soit triée.

`tri_insere([],[]).`

`tri_insere([X:Queue],Triée) :-`
 `tri_insere(Queue,QueueTriée),` (trie la queue)
 `insere(X,QueueTriée,Triée).` (Insère X correctement).

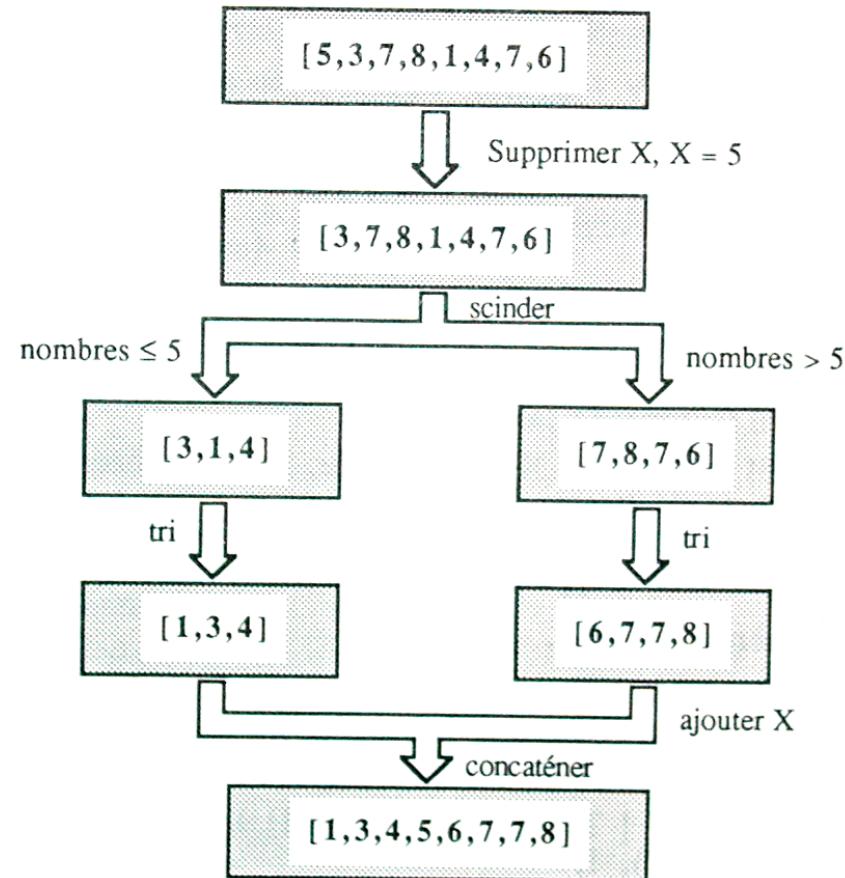
`insere(X,[Y:triée],[Y:triée]) :-`
 `X>Y,`

`insere(X,triée,triée1).`

`insere(X,Triée,[X:triée]).`



Tri de listes – Troisième solution



Tri de listes – Troisième solution

```
trirapide([], []).  
  
trirapide([X|Reste], Triée) :-  
    scinde(X, Reste, Petite, Grande),  
    trirapide(Petite, PetiteTriée),  
    trirapide(Grande, GrandeTriée),  
    conc(PetiteTriée, [X|GrandeTriée], Triée).  
  
scinde(X, [], [], []).  
  
scinde(X, [Y|Reste], [Y|Petite], Grande) :-  
    pg(X, Y), !,  
    scinde(X, Reste, Petite, Grande).  
  
scinde(X, [Y|Reste], Petite, [Y|Grande]) :-  
    scinde(X, Reste, Petite, Grande).  
  
conc([], L, L).  
  
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```



Eliminer les doublons d'une liste.

Soit la clause : `doublon(L1,L2)`, avec `L1` la liste avec des doublons, et `L2` la liste sans doublon.

Principe :

- (1) Une liste avec un seul élément n'a pas de doublon (condition d'arrêt).
- (2) Soit une liste `L`. Isoler sa tête `X` de sa queue `Lq`.
 - si `X` appartient à la liste `Lq`, renvoyer la liste `Lq` sans la tête.
 - Si `X` n'appartient pas à la queue, renvoyer la liste complète (garder la tête).

Soit en prolog (voir précédemment la programmation de la clause `appart`) :

<code>doublon([X],[X]).</code>	(une liste avec 1 élément n'a pas de doublon).
<code>doublon([X:L1],L2) :- appart(X,L2), doublon(L1,L2).</code>	(<code>X</code> est un doublon, continuer en coupant la tête).
<code>doublon([X:L1],L2) :- doublon(L1,L3), L2=[X:L3].</code>	(sinon <code>X</code> n'est pas un doublon, garder la tête).

