Paradigmes de programmation

Une introduction

Olivier PORTE

CNRS

Mai 2012

Sommaire

Sommaire

Objectifs de la présentation

- Approcher quelques fondements de l'informatique
  - Survol rapide de quelques grands principes
  - Positionner les limites du développement
- Améliorer le niveau d'invariance des connaissances dans un environnement fortement évolutif
- Répondre à quelques questions : les langages sont-ils équivalents, peut-on traiter tous les problèmes, etc.
- Expliciter quelques paradigmes de programmation et les illustrer
- Fournir un éclairage un peu moins "traditionnel" de la programmation
- Se projeter sur quelques enjeux actuels et à venir

Sommaire

Références bibliographiques

- Bibliographie abondante et souvent ardue
- Sujets de niveau recherche (on touche les fondements de la discipline !)
- Difficile de traiter un tel sujet en peu de temps : il faut s'investir via la bibliographie pour progresser

Bibliographie I

[CAR] Alain CARDON & Christian CHARRAS & Daniel KROB, Un retour aux origines du calculable - La thèse de CHURCH/POST, Groupe Informatique de l'IREM de Rouen

[VER11] Didier Verna, Approches Fonctionnelles de la Programmation - Introduction, Version EPITA, 31 janvier 2011

[JEZ09] Loïg JEZEQUEL, Paradigmes de programmation, ENS Cachan Bretagne, 16 novembre 2009

[ALO03] Allouch Bachir & Trantoul Gilles, Les nouvelles formes de programmation, juin 2003

[LOO04] Pierre De Loor, Autres langages de l'IA - Module Programmation Logique, ENIB, 2004

[BON08] D.Bonnay, Logique et calcul, ParisX & IHPST-DEC, Cogmaster 2008-2009

[HAM] N. Hameurlain, Introduction : Langages de Programmation, LIN 2

[JOU06] Pierre Jouvelot, Enseignement spécialisé Informatique fondamentale (S1214), Centre de recherche en informatique - École des mines de Paris, 2006

[MEN] Tom Mens & Johan Brichau, Paradigmes de Programmation, Université de Mons-Hainaut Service de Génie Logiciel & Université catholique de Louvain Ingénierie Informatique, Cogmaster 2008-2009

[PIG07] C. Piguet & H. Hügli, La programmation, 18/01/2007

[JAU] Mathieu Jaume & Frédéric Peschanski, Introduction à la sémantique des langages de programmation, LIP6 -Université Paris 6

[HUT] Guillaume Hutzler, Informatique Générale, Laboratoire IBISC (Informatique Biologie Intégrative et Systèmes Complexes), Cours Dokeos

[PIG04] C. Piguet & H. Hügli, Du zéro à l'ordinateur - Une brève histoire du calcul, Presses Polytechniques et Universitaires Romandes, 2004

[PEN97] Roger Penrose, L'esprit, l'ordinateur et les lois de la physique, InterEditions, 12/1997

[CAS07] Pierre Cassous-Noguès, Les démons de Gödel - Logique et folie, Éditions du Seuil, Septembre 2007

Bibliographie II

[DOX10] Apóstolos K. Doxiàdis (Auteur) & Christos Papadimitriou (Auteur)& Alecos Papadatos (Illustrations) & Annie Di Donna, Logicomix, Éditions Vuibert, 2010

[DEL06] Jean-Paul Delahaye, Complexités - Aux limites des mathématiques et de l'informatique, Éditions Belin Pour la science, 2006

[CAI06] Jerry Cain, Programming paradigms, Stanford Engineering, CS107, Stanford University, 2008

[BEL09] Jean-Pierre Belna, Histoire de la théorie des ensembles, L'esprit des sciences, Ellipses, 2009

[DAN11] Jean Daniel, Les mathématiques. Les plus grands textes d'Euclide à Gödel et Bourbaki, collection dirigée par Jean Daniel, L'anthologie du savoir, Le Nouvel Observateur et CNRS Éditions, 2011

[DES05] Jean-Marc Deshouillers, Les théorèmes de Gödel : fin d'un espoir ?, Université Bordeaux Segalen, Canal U, Vidéothèque Numérique de l'Enseignement Supérieur, saison 2005-2006

[BAC89] R. Backhouse, Construction et vérification de programmes, Masson, Paris, 1989

[CAR90] Christian Carrez, Des structures aux bases de données, Dunod, 1990

Sommaire

Notion de "Paradigme"

- Un paradigme est un modèle ou "patron" de pensée
- Cela permet de définir un ensemble de règles ou de concepts permettant d'appréhender un cadre particulier
- Ces concepts, adaptés à un domaine précis, permettent de voir un aspect de la réalité
- Ainsi, chaque science dispose d'un ou plusieurs paradigmes adaptés à son sujet d'étude
- En conséquence, la vision de chaque science sur un problème particulier est différente voire hors du scope pour certaines (c'est hors de son champ de perception)
- Exemple : l'astrologie n'a aucun sens pour la Physique moderne mais peut être sujet d'étude pour la Psychologie et/ou la Sociologie

Notion de "Paradigme"

- Une des motivations de l'interdisciplinarité se trouve ici : la vision d'une même problématique avec des points de vue complémentaires est potentiellement une source de progrès
- De nombreux exemples de changement de paradigme existent en sciences
- On peut citer : la passage du Géocentrisme à l'Héliocentrisme ou le passage de la mécanique classique à la mécanique quantique
- 27 avril 1900, Lord Kelvin : "La connaissance en physique est semblable à un grand ciel bleu, à l'horizon duquel subsiste seulement deux petits nuages ..."

Notion de "Paradigme"

- Deux petits nuages dont :
  - l'un, l'expérience de Michelson et Morley ... donnera la relativité
  - l'autre, le rayonnement du "corps noir" ... donnera la mécanique quantique
- ***En conclusion*** :
  - choisir le bon paradigme selon le problème à considérer
  - ne surtout pas négliger les petits détails qui peuvent être des prémisses aux changements de paradigmes !

Notion de "Programmation"

- Vision macroscopique : faire le lien entre des données d'entrée et de sortie
- Oui, mais encore ...
- Traiter des phrases :
  - Vérifier que les phrases sont correctes : travail sur la syntaxe
  - Vérifier que les phrases ont un sens : travail sur la sémantique
- Deux mécanismes :
  - Génération de phrases : travail sur les grammaires
  - Reconnaissance de phrases : automates, compilateurs.

Notion de "Programmation"

- Programmer, c'est donc :
  - Observer le réel
  - En construire un modèle en choisissant une vision ou domaine d'interprétation (application d'un paradigme)
  - Traduire ce modèle pour une machine via un langage dit de "programmation"

Notion de "Programmation"

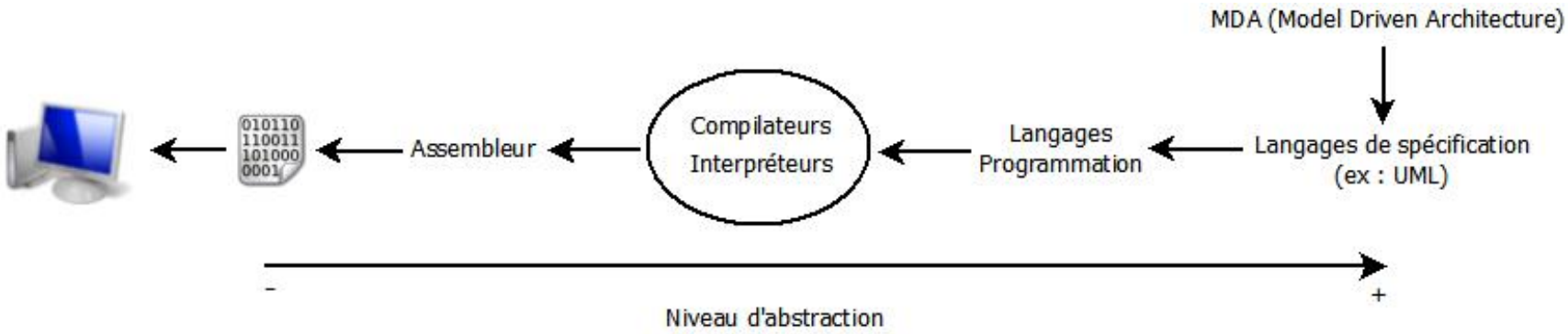


Figure: Cycle programmation

Notion de "Programmation"

- In fine, un ordinateur ne connait que le format binaire ...
- Tout ce que nous allons évoquer dans la suite se traduira donc en instructions pour un/des processeurs
- A l'origine, il n'y avait que le binaire, puis l'assembleur, des "macros" pour l'assembleur ...
- ... et des langages sont apparus pour faciliter la vie du développeur et regrouper ces "macros" dans des instructions de plus haut niveau
- Par la suite, sous la double contrainte de la complexité des sujets à traiter et de la nécessaire maintenance des développements, différents paradigmes et langages associés ont été introduits

Notion de "Programmation"

- Mais ces paradigmes et les propriétés relatives à la programmation ne datent pas du premier ordinateur ...
- En fait, en tant que "mathématique appliquée", la programmation est issue de grandes problématiques ayant émergées au début du 20^{e} siècle
- En effet, "programmer" consiste à formaliser un modèle du réel et à le rendre exécutable
- ***Programmer, c'est donc axiomatiser un problème !***

Notion de "Programmation"

- Edsger DIJKSTRA (1930-2002)- Mathématicien et physicien néerlandais
- Prix Turing 1972 pour ses nombreuses contributions à l'informatique (PCC, etc.)

Figure: E.W Dijkstra (source Wikipédia)

- ***"La programmation est la branche la plus difficile des mathématiques appliquées"***

Programmer, c'est axiomatiser des problèmes

- Le niveau de difficulté des problèmes s'échelonne de "simple à compliqué"
- Attention : il n'existe pas de solutions simples aux problèmes compliqués
  - Si cela semble être le cas, c'est que le problème était en réalité mal posé
  - Seule "la complexité tue la complexité" (question d'entropie)
- Un exemple : le réseau
  - Problèmes complexes à résoudre : gestion de congestion, priorisation de flux, etc.
  - Fin des années 1990 : ATM (Asynchronous Transfert Mode) - Protocole complexe ...
  - Années 2000 : règne d'Ethernet et IP ... mais les problèmes subsistent, camouflés par une stratégie "d'overprovisionning" des liaisons, de BGP/MPLS, etc. dont la complexité et le coût n'ont parfois rien à envier à ATM
- "On a rien sans rien ..."

Programmer, c'est axiomatiser des problèmes

- Lorsqu'un problème est abordé, deux questions se posent :
  - Est-ce que le problème à une solution : notion de calculabilité
  - Est-ce que la solution peut être trouvée dans un délai raisonnable : notion de complexité
- De manière plus académique :
  - Calculabilité : qu'est-ce qu'un algorithme "effectif" et quels sont les problèmes sur lesquels il s'applique ?
  - Complexité : qu'est-ce qu'un algorithme efficace et comment le mesurer ?
- D'un côté le "théoriquement possible" et de l'autre le "pratiquement possible"

Sommaire

Des questions de fond et une ambition pour la science

- 1900 : 2e congrès des mathématiciens (tous les 4 ans)
  - David Hilbert présente 23 problèmes majeurs à résoudre en mathématique
  - Le 10ième problème annonce ce qui deviendra un enjeu majeur : "peut-on décider, de façon mécanique, si une équation Diophantienne a une solution ?"
  - Exemple d'équation Diophantienne : ax+by = c (Wikipédia)

Figure: D. Hilbert (1862-1943) (source bibm@th.net)

Des questions de fond et une ambition pour la science

- 1928 : Congrès des mathématiciens
- David Hilbert évoque 3 questions (parmi d'autres) :
  - Les mathématiques sont-elles complètes ? C-à-d, tout énoncé mathématique peut-il être soit prouvé soit réfuté ?
  - Les mathématiques sont-elles cohérentes (ou "consistantes" ou "non contradictoires") ? C-à-d, peut-on être sûrs que des raisonnements valides ne conduiront pas à des absurdités ou des contradictions (un énoncé et son contraire) ?
  - Les mathématiques sont-elles décidables ? C-à-d, existe-t-il un algorithme pouvant dire de n'importe quel énoncé mathématique s'il est vrai ou faux ? (Entscheidungsproblem)
- Ambition de Hilbert affichée dès 1900 : construire un système axiomatique formel (cohérent et complet) pouvant décrire toutes les mathématiques et, in fine, déduire mécaniquement l'ensemble des théorèmes
- **Commentaire de Henri Poincaré** : "Mr Hilbert pense que les mathématiques sont comme la machine à saucisses de Chicago : on y introduit des axiomes et des cochons, et en sortent des saucisses et des théorèmes"

Une ambition contrariée ... l'incomplétude de Gödel

- 1931 : Kurt GÖDEL répond aux deux premières questions de Hilbert
  - 1 : Tout système formel suffisamment puissant est soit incohérent soit incomplet
  - 2 : Si un système d'axiomes est cohérent, cette cohérence ne peut être prouvée en n'utilisant que les axiomes du système en question
- Autrement dit, aucun système de preuve n'est complet et il n'est pas possible de prouver un système de preuve avec ce même système de preuve (autoréference) ... dans tout système de preuve (contenant la logique et l'arithmétique), il existe des propositions vraies que l'on ne peut pas prouver (tout problème de décision ne peut être résolu avec la logique)
- Cela contre-carre sérieusement l'ambition du programme de Hilbert visant à unifier les mathématiques dans un seul et unique système cohérent d'où découlerait l'ensemble des mathématiques, de manière mécanique

Un mot sur les théorèmes d'incomplétude de GÖDEL

- 1931 : Kurt GÖDEL (1906-1978) - Mathématicien et logicien Austro-Américain

Figure: K. Gödel (source jml.fr)

Un mot sur les théorèmes d'incomplétude de GÖDEL

- Soit T une théorie complète et cohérente (axiomes et règles) permettant de déterminer la vérité ou la fausseté d'une proposition
- Soit P une proposition telle que : P = $\neg\exists x \setminus [T \Rightarrow P]$ (c-à-d "T ne dira jamais que P est vraie"). Proposition sémantiquement vraie mais singulièrement paradoxale ...
- Car, à la question : est-ce que P est vraie par rapport à T ?
- Il existe deux réponses possibles :
  - Si T dit que P est vraie, alors P est fausse ... T étant en principe complète et cohérente, T ne dira donc jamais P
  - Si T ne dit pas que P est vraie, alors P est vraie ... mais contredit T
- Conclusion : P est indécidable par rapport à T

Un mot sur les théorèmes d'incomplétude de GÖDEL

- C'est le paradoxe du menteur : « un homme déclare "Je mens". Si c'est vrai, c'est faux. Si c'est faux, c'est vrai. » (Wikipédia)
- Il a donc été prouvé qu'il n'est pas possible de tout prouver relativement à un système donné
- Et, en plus, il n'y a pas d'échappatoires : inclure des axiomes nouveaux, etc.
- Par rapport à la question de Hilbert, "décider de façon mécanique" n'est donc pas possible.

Un mot sur les théorèmes d'incomplétude de GÖDEL

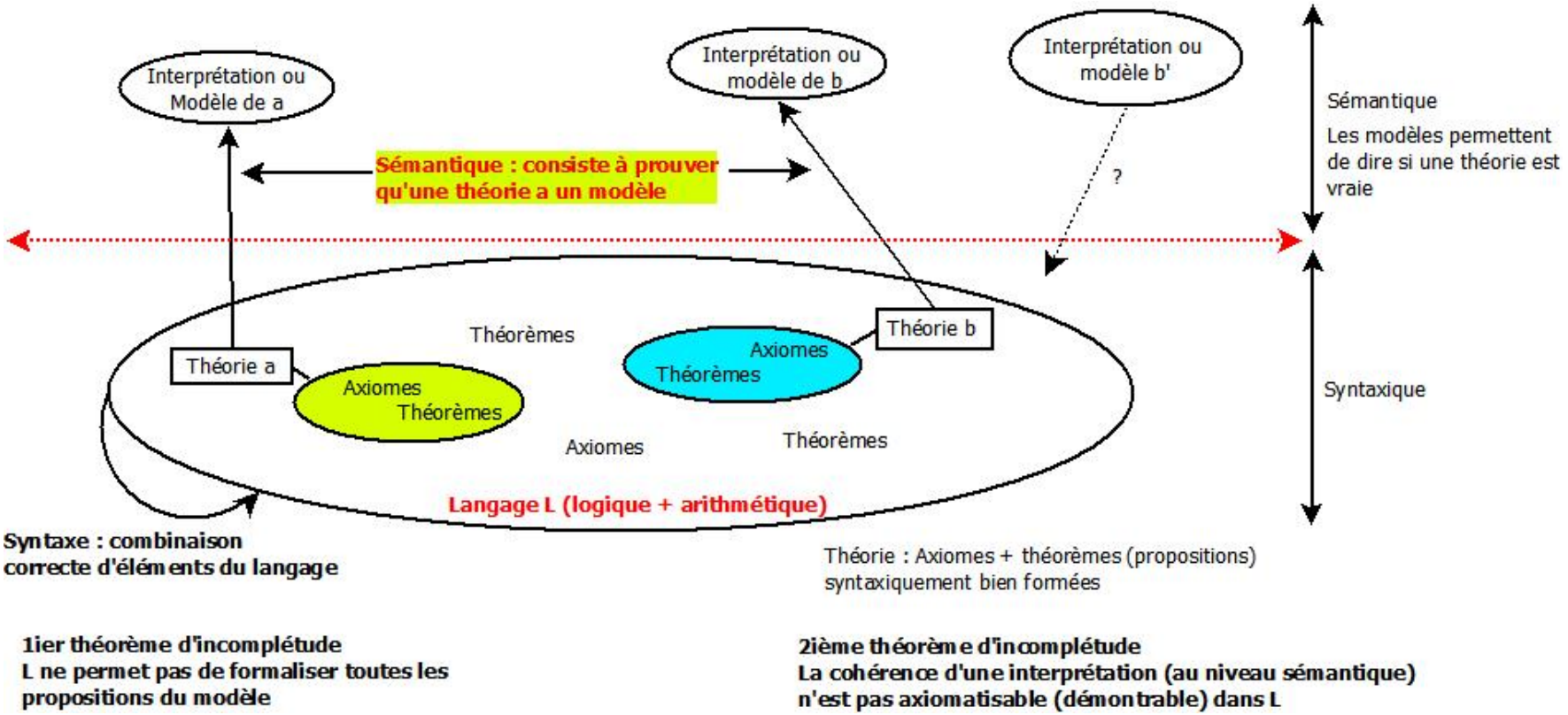


Figure: Théorèmes d'incomplétude

Une ambition contrariée ... Turing et l'Entscheidungsproblem

- 1936 : Alan TURING résout le problème de l'arrêt
- Il restait en effet une question en suspend suite aux théorèmes de Gödel : existe-t-il une procédure mécanique permettant de savoir si une assertion est vraie ou fausse (3^e question de Hilbert) ?
- Pour prouver l'impossibilité de cette procédure, A. Turing :
  - construisit d'abord un modèle formel de calculateur (Machine de Turing : MT)
  - prouva ensuite que la MT ne peut résoudre certains problèmes dont le problème de l'arrêt (Halting Problem)

Figure: A. Turing (1912-1954) (source wired.com)

Une ambition contrariée ... Turing et l'Entscheidungsproblem

- "Une machine de Turing est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire, créé par Alan Turing en vue de donner une définition précise au concept d'algorithme ou « procédure mécanique »" (Wikipédia)
- Ce modèle abstrait est supposé à "ressources infinies", avec un "ruban" servant de mémoire et dont la longueur n'a potentiellement pas de limite
- On peut encoder une machine de Turing sous la forme d'une chaîne de caractères
- Une MT peut simuler le comportement d'une MT : "machine de Turing universelle"
- C'est avec cette MT universelle que le problème de l'arrêt peut être modélisé (on demande à une MT de "se prononcer" sur l'arrêt d'une autre MT)

Une ambition contrariée ... Turing et l'Entscheidungsproblem

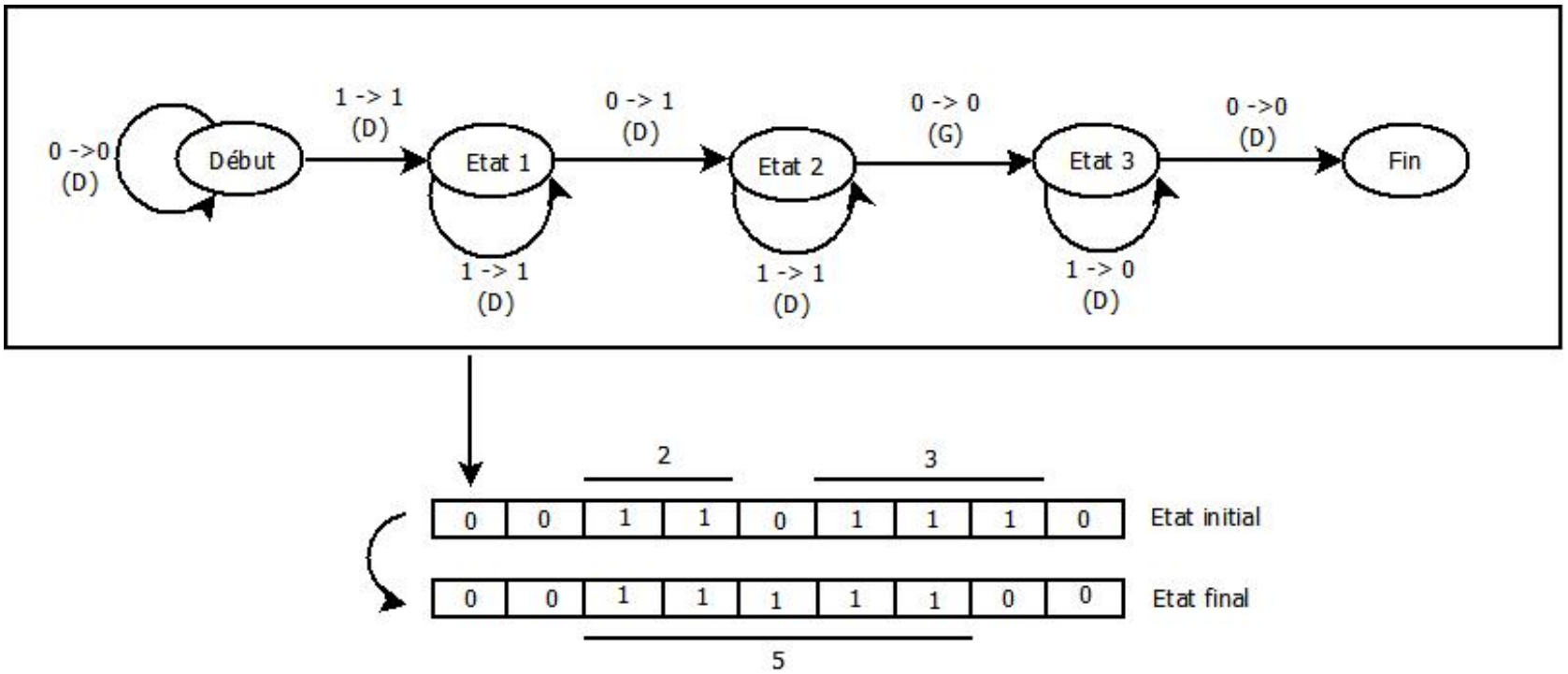


Figure: Addition binaire avec une MT

Une ambition contrariée ... Turing et l'Entscheidungsproblem

- Une approche de problème de l'arrêt (par l'absurde) :
  - Soit Halt(P) un programme qui répond True si P se termine et False sinon
- Soit Absurde le programme suivant :

```
Procedure Absurde
Début
        Si (Halt(Absurde)) alors
           Tant que "TRUE" faire Ecrire ("TRUE")
        FinSi
Fin
```

Une ambition contrariée ... Turing et l'Entscheidungsproblem

- Le programme "Absurde" ne permet pas d'avoir de réponse ... le problème de l'Arrêt est indécidable
- Cela a des conséquences pratiques dans nos programmes et outils
- Exemple : nettoyage de la mémoire utilisée par un programme : oui ... mais quand ? Lorsque le programme (procédure, fonction) est fini ...
- La démonstration de Turing acheva le travail commencé par Gödel
- Cela mis un terme définitif à l'ambition du programme de Hilbert visant à construire un système formel universel qui soit cohérent, complet et ayant pour objet de permettre une démonstration mécanique de théorèmes

Architecture de von Neumann

- A noter que les MT préfigurent l'architecture de Von Neumann proposée dans les années 40 ... et l'architecture de nos ordinateurs actuels !

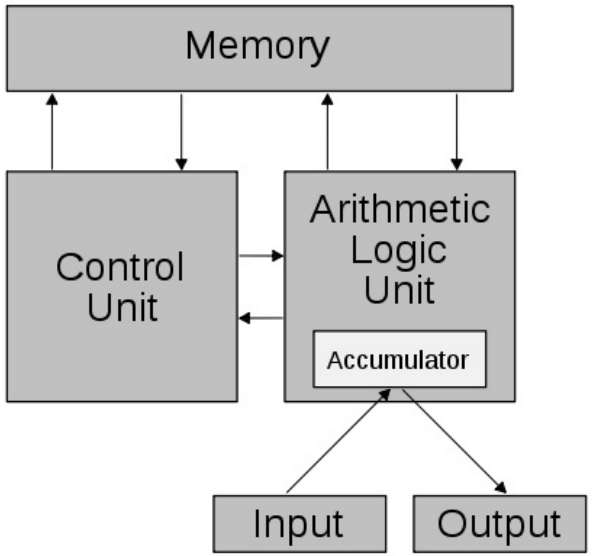


Figure: Architecture de Von Neuman (source Wikipedia)

Un mot sur la thèse de Turing-Church

- 1936 : Thèse de TURING-CHURCH (Thèse car non encore prouvé formellement ...)
  - Tout problème de calcul basé sur une procédure algorithmique peut être résolu par une MT
  - Toute procédure algorithmique peut être ramenée (est équivalente à) une MT
  - Tout problème solvable sur un ordinateur est aussi solvable sur une MT

En conclusion

- "Programmer, c'est axiomatiser un problème"
- Axiomatiser fait référence à Gödel, Turing, etc.
- Les informaticiens doivent donc travailler avec :
  - un système soit incohérent soit incomplet
  - dont il n'est pas possible de prouver automatiquement s'il s'arrête
- Comment travailler dans ces conditions ? ... voyons la suite ;-)

Sommaire

Les formalismes

- Plusieurs formalismes ont été/sont introduits pour tenter de résoudre les questions majeures posées en mathématique (40 à 50 formalismes différents)
  - Machine de Turing
  - Lambda calcul
  - Systèmes de réécriture : grammaires, Types Abstraits de Données (TAD)
  - Systèmes de contraintes : résolution par satisfaction de "buts" dans un système logique
  - Etc.
- La thèse de Turing-Church met en évidence l'équivalence de l'ensemble de ces paradigmes

Les formalismes

- Conséquences importantes :
  - tous les langages de programmation sont équivalents à une MT
  - un nouveau langage ne peut pas exprimer plus que ceux déjà existants
- Certains problèmes sont plus faciles à résoudre avec certains langages (plus expressifs ou plus spécialisés)
- Corolaire : inutile de disserter sur les mérites comparés de tel ou tel langage : il faut choisir le plus adapté au problème considéré (surtout pour les applications critiques ou liées à la sécurité (IGC, etc.))
- Exemples de paradigmes : objets concurrents/programmation concurrente, multi-agents, aspects, événementiel, ...

Les formalismes et les paradigmes

- Les formalismes explorés ont donné de nombreux paradigmes en programmation
- Seuls quelques uns jouent un rôle fondamental car se retrouvent dans de nombreux langages :
  - Machine de Turing : paradigme "impératif"
  - Lambda Calcul : paradigme "fonctionnel"
  - Système de contraintes : paradigme "logique"
  - Types Abstraits de Données : paradigme "objet"
- ***RAPPEL : ces paradigmes sont tous équivalents*** (thèse de Turing-Church)

Le choix d'un langage

- Le choix d'un langage doit donc se faire selon la nature du problème à appréhender
- Cela simplifie le formalisme, mais, dans l'absolu, tout langage convient à tout
- Il existe malgré tout des critères pratiques pour choisir le langage adapté à un problème :
  - capacité à mettre en œuvre plusieurs paradigmes
  - facilité d'apprentissage
  - syntaxe adaptée au contexte
  - compilé/interprété, compilateurs optimisés selon l'environnement
  - normalisation (JSR pour Java, ISO/CEI 8652 pour ADA (AFNOR NZ 65700), etc.)
  - communauté active
  - notoriété et capacité à trouver des développeurs
  - outils associés : éditeurs, librairies, etc.
  - etc.

Les langages et les paradigmes

- Paradigme impératif
  - historiquement le premier paradigme implémenté
  - suite d'instructions qui modifient la mémoire centrale et les registres du processeur
  - un programme peut se traduire par une machine à états : états successifs de la mémoire
  - ce type de programme est conceptuellement proche de l'architecture machine de von Neumann
  - le programmeur doit connaitre à tout instant l'état de la mémoire (très vrai en C !)
  - principe : application d'instructions (boucles, tests conditionnels, sauts, etc.)

Les langages et les paradigmes

- Paradigme fonctionnel
  - que des fonctions
  - pas de gestion directe de la mémoire, sauf pour des raisons d'efficacité (on parle alors de langages fonctionnels impurs)
  - application d'une fonction qui retourne un résultat (comme en math)
  - utilisation massive car naturelle de la récursivité
  - principe : application et évaluation de fonctions

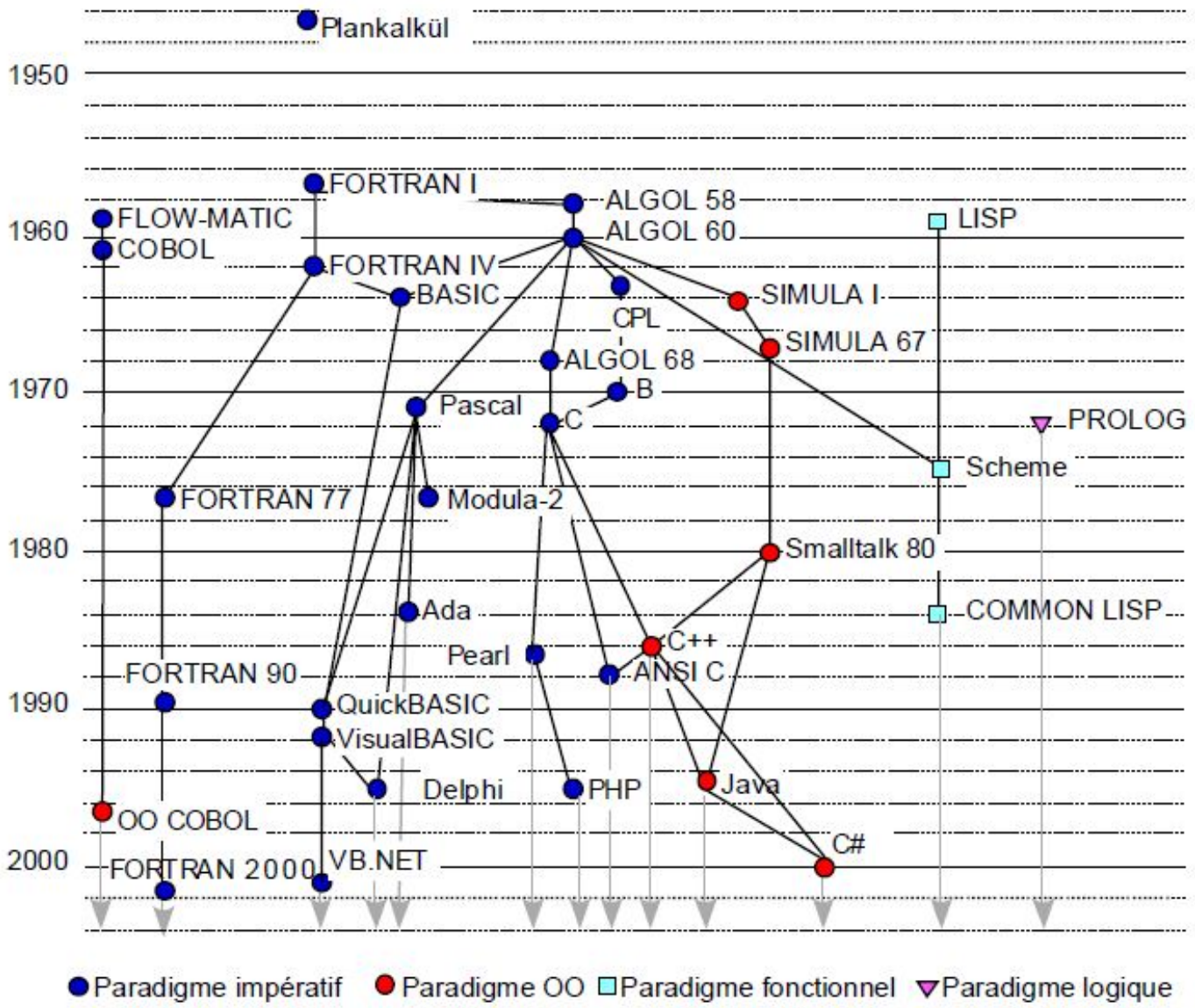
Les langages et les paradigmes

- Paradigme logique
  - fondé sur la logique mathématique
  - déclaration de faits et de règles (axiomes et règles)
  - résolution en "posant une question", c-à-d en déclarant un but qui sera évalué en se basant sur les axiomes et les règles définis
  - principe : unification (substitution de variables pour rendre identiques des prédicats), chainage arrière (on part du but pour trouver les hypothèses via les règles et axiomes)

Les langages et les paradigmes

- Paradigme objet
  - moins fondamental que les précédents mais très souvent évoqué et utilisé
  - fondé sur les Types Abstraits de données (TAD) ... donc rien de très nouveau
  - encapsulation, polymorphisme, héritage
  - principe : appel des méthodes des objets
- Point de vue de E. DIJKSTRA : "La programmation par objets est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie"

La "filiation" des langages (C. Piguet et H. Hügli [PIG07])



Notions importantes pour la contruction d'un programme

- Il est désormais possible de préciser :
  - la sémantique d'un programme
  - les preuves de programmes
  - les Types Abstraits de Données (TAD)
- Pourquoi revenir sur le sujet ?
  - pour voir comment construire des programmes plus sûrs
  - illustrer cela à travers des exemples

Petite synthèse

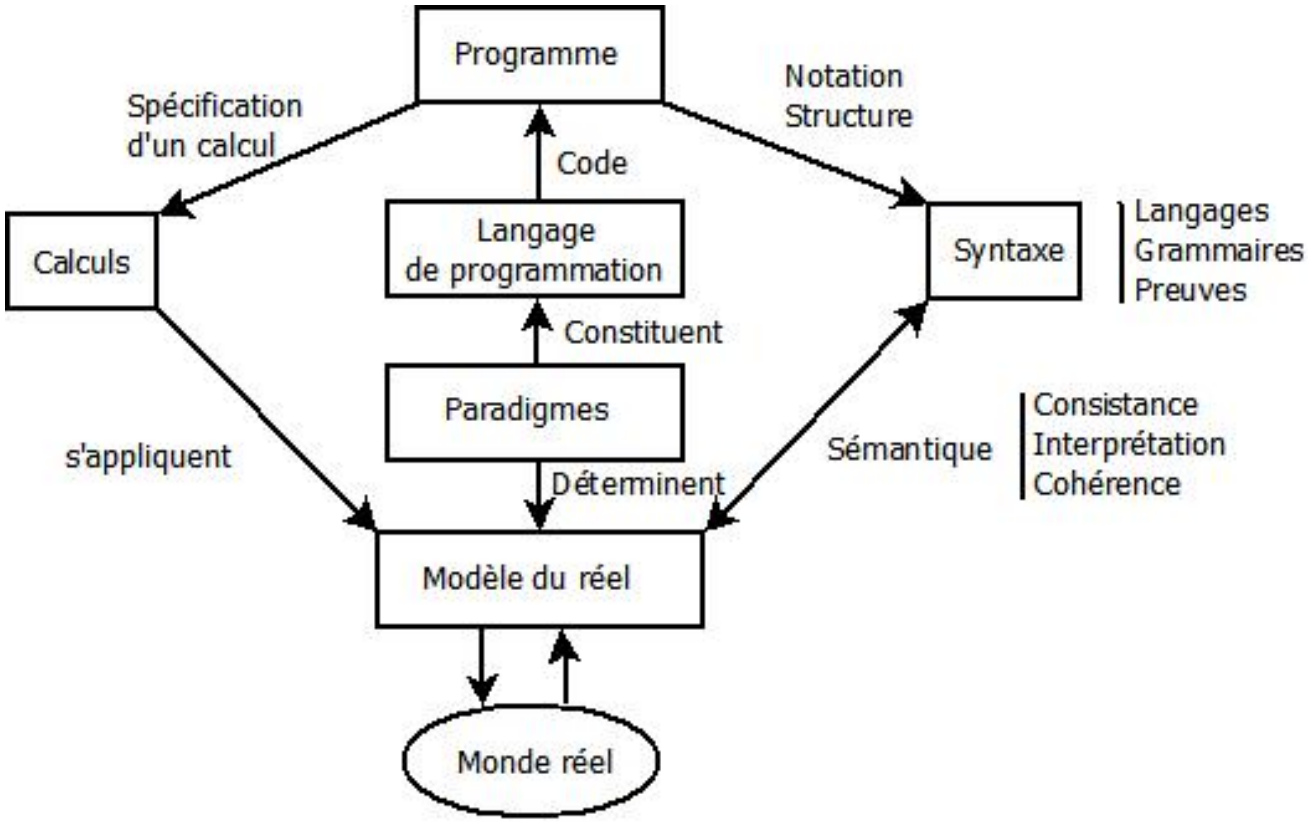


Figure: Monde réel, modèle et syntaxe d'un programme

La sémantique d'un programme : approche pratique

- Qu'est-ce que la sémantique d'un programme ?
- Programme = ensemble d'axiomes
- Exemple en Prolog :

```
1  /* Faits : clause d'un seul atome */
2  r.
3  q.
4  /* Procedures ou regles */
5  p :- q,r.
6  p :- s.
```

- Chaque ligne est une clause et $\mathbf{P} = \sum_{i=1}^{n} Clause_i$

La sémantique d'un programme : approche pratique

- Sur cet exemple, comment déterminer le sens du programme ?
- Cette propriété s'appelle "dénotation". Notons-là D.
- Au début, D = $\varnothing$
- On ajoute ensuite les faits : D = $\{r, q\}$
- On choisit ensuite une règle. Ex : p :- q,r. Si toutes les conditions sont vérifiées, on ajoute p à la dénotation D.
- Ici, D = $\{p, q, r\}$
- On itère sur l'ensemble des règles et, à la fin du processus, D représente la sémantique de P.
- Autrement dit : "P calcule D" ou "P a pour sens D"
- Remarque : langage C et grammaire du C

La sémantique d'un programme : approche pratique

- Dans l'exemple précédent, on calcule le "point fixe" du programme pour trouver le sens de P
- Comment se détermine ce point fixe ?
- Un peu de vocabulaire préalable :
  - un sous-ensemble de P s'appelle une interprétation de Herbrand
  - $\{p, q, r, s\}$ représente la base de Herbrand du programme
- Soit I une interprétation de Herbrand et et T_p une fonction de transition du programme P (typiquement une règle)
- On calcule $T_p(\varnothing)$, $T_p(T_p(\varnothing))$, ... dans l'exemple, on obtient $\{r, q, p\}$ soit $T_p^2(\varnothing)$
- $Denotation_p = \bigcup_{n=0}^{\infty} T_p^n(\varnothing)$

La sémantique d'un programme : approche pratique

- Le moment intéressant arrive lorsque $T_p(T_p...(T_p(\varnothing))...)$ n'apporte plus rien
- C'est la dénotation "point fixe"
- Il existe une dénotation procédurale, une dénotation logique, etc.
- Par exemple, pour la dénotation procédurale, p :- q,r. devient :

```
1 Procedure P
2 Debut
3 q;
4 r;
5 Fin
```

- Au total : $Den_{Pf} = Den_{Log} = Den_{Proc}$

Les preuves de programmes

- Il existe plusieurs approches sur le sujet
- Ces approches sont basées sur l'isomorphisme de CURRY-HOWARD
  - une preuve mathématique est un programme
  - Preuves(Axiomes, théorèmes) $\equiv$ Développement(Spécifications, programmes)
- Les plus connues sont celle de HOARE [décrit dans Unified theories of programming - juillet 1994] et celle de DIJKSTRA avec les "weakest predicate" (WP)
- L'approche de Hoare est basée sur l'utilisation et la formalisation de règles d'inférence
- L'approche de Dijkstra est basée sur les notion de pré et post conditions ainsi que de "condition la plus faible" (WP)

Les preuves de programmes : approche de DIJKSTRA

- Quelques éléments de base :
  - on part de : $\{P\}$ S $\{Q\}$ où
    - $\{P\}$ correspond aux pré-conditions
    - $\{Q\}$ correspond aux post-conditions
    - S représente le code (suite d'instruction)
- Dans ce contexte, un "weakest predicate" est la pré-condition la plus faible permettant de décrire les états (mémoire) initiaux garantissant que l'exécution de S aboutira à l'état Q
- Les pré et post-conditions sont des assertions, c-à-d un formalisme en logique du 1er ordre appliqué sur des variables (logique du premier ordre : logique des propositions + quantificateurs sur les variables)

Les preuves de programmes : approche de DIJKSTRA

- Des exemples :
  - l'affectation : i = i - 1
    - wp(i=i-1,$\{i = 0\}$) $\equiv$ (i=0)[i/i-1] $\equiv$ i-1=0 $\equiv$ i=1
  - une permutation de variables : temp = x ; x = y ; y = temp
    - wp(temp=x ;x=y ;y=temp,$\{x = y_0 \wedge y = x_0\}$)
    - wp(temp=x ;x=y,$\{wp(y = temp, \{x = y_0 \wedge y = x_0\})\}$)
    - wp(temp=x ;x=y,$\{x = y_0 \wedge temp = x_0\}$)
    - wp(temp=x,$\{wp(x = y, \{x = y_0 \wedge temp = x_0\})\}$)
    - wp(temp=x,$\{y = y_0 \wedge temp = x_0\}$) $\equiv \{y = y_0 \wedge x = x_0\}$
    - Si on a $\{y = y_0 \wedge x = x_0\}$ et que l'on exécute la permutation, on obtient $\{x = y_0 \wedge y = x_0\}$ au final
    - Heureusement ...

Les preuves de programmes : approche de DIJKSTRA

- Des exemples (suite) :
  - test conditionnel : Si (x>5) alors y=3 sinon (y=1 ;x=x-1) FinSI avec comme post-condition $\{x > y\}$
    - (x>5 $\Rightarrow$ wp(y=3,$\{x > y\}$)) $\wedge$ (x$\leqslant$5 $\Rightarrow$ wp(y=1 ;x=x-1,),$\{x > y\}$))
    - $(x > 5 \Rightarrow x > 3) \wedge (x \leqslant 5 \Rightarrow wp(y = 1, \{wp(x = x - 1), \{x > y\}\}))$
    - $(x > 3) \wedge (x \leqslant 5 \Rightarrow wp(y = 1, \{x > y + 1, \}))$
    - $(x > 3) \wedge (x \leqslant 5 \Rightarrow x > 2) \equiv (x > 3) \wedge (x > 2) \equiv (x > 2)$
  - Instructions "Tant que"
    - montrer un exemple serait plus long : se reporter à la bibliographie
    - le principe dans le cas $\{P\}\ WhileBdoS\ \{Q\}$ est de :
    - considérer le cas ne non entrée dans la boucle : $P0 = \neg B \wedge Q$
    - considérer ensuite les itérations : $B \wedge wp(S, P0)$, etc. et plus généralement $P_k = B \wedge wp(S, P_{k-1})$
    - en pratique, par récurrence,on voit apparaitre les variants et les invariants de boucle qui permettent d'obtenir les pré-conditions

Les preuves de programmes : approche de DIJKSTRA

- Ces principes sont essentiels en Génie Logiciel
- Ils permettent de :
  - garantir la correction des programmes
  - fournir une documentation et un niveau de commentaires qui soient explicites
- Des implémentations pratiques existent pour certains langages (voir la suite)

Les Types Abstraits de Données (TAD)

- Les TAD font partie de l'approche algébrique
- Ils permettent de décrire un système par l'ensemble des opérations qui s'appliquent à ce système
- Cette approche permet de saisir le comportement du système
- Concrètement, la syntaxe des TAD s'appuie sur :
  - des objets
  - des opérations sur ces objets
  - les propriétés des opérations sur les objets
- Les données et les opérations fournissent la signature
- Les propriétés sont définies par des axiomes et fournissent la sémantique

Les Types Abstraits de Données (TAD)

- Plus précisément encore :
  - un TAD se représente ainsi : <Sig,C> où "Sig" est la signature (données+opérations) et "C" l'ensemble des axiomes
  - la signature se décompose en : $Sig =< S, \Omega, \Pi >$ ou "S" définit le type, "Ω" l'ensemble des opérations et "Π" l'ensemble des pré-conditions
- Les opérations sont typées : pour toute opération, le type (ensemble définit de valeurs que peuvent prendre les variables) des arguments de chaque opération est fixé

Les Types Abstraits de Données (TAD)

- Voici un exemple concret de TAD :

Type :	Pile
Paramètre :	T
Utilise :	booléens
Opérations :	créer : $\rightarrow$ Pile[T]
	vide : Pile[T] $\rightarrow$ bool
	empiler : T,Pile[T] $\rightarrow$ Pile[T]
	sommet : Pile[T] $\nrightarrow$ T
	dépiler : Pile[T] $\nrightarrow$ Pile[T]
Préconditions :	dépiler : $\neg$ vide(Pile[T])
	sommet : $\neg$ vide(Pile[T])
Variables :	p : Pile et t : T
Axiomes :	vide(créer()) = vrai
	$\neg$ vide(empiler(t,p)) = vrai
	sommet(empiler(t,p)) = t
	dépiler(empiler(t,p)) = p

Les Types Abstraits de Données (TAD)

- Quelques remarques :
  - il est possible de mettre en œuvre une hiérarchie entre types : paragraphe "utilise"
  - il est possible de définir un type générique via "Paramètre"
  - il est possible de distinguer 3 types d'opérations :
    - les constructeurs : seule la partie droite comprend le type
    - les transformateurs : utilisent le type en argument et en résultat
    - les observateurs : le type n'apparaît qu'en résultat
  - les opérateurs ne sont pas obligatoirement définis pour toute les valeurs des arguments. Les restrictions se retrouvent alors en pré-conditions
- La partie axiomes pose deux problèmes classiques : cohérence et complétude
- Partie à laquelle s'ajoute la question de la consistance : y-a-t-il suffisamment d'axiomes pour décrire parfaitement le TAD ?

Les Types Abstraits de Données (TAD)

- L'approche TAD est à l'origine de l'approche objets
- Cette approche est utilisé dans des langages comme ADA où la notion de "spécification" correspond à la notion de signature (sans les axiomes)
- Hormis ADA, d'autres langages implémentent aussi cette notion, en incluant, soit nativement, soit par ajouts, la dimension "axiomes"

Un mot sur la complexité

- Il existe deux types de complexité :
  - La complexité des problèmes
  - La complexité des programmes (nombre d'opérations élémentaires à effectuer sur un jeu de données)

La complexité des problèmes

- Les problèmes sont "classifiables" selon leur niveau de difficulté
- Cette classification est indépendante des moyens mis en œuvre pour les résoudre
- Sommairement, il existe :
  - des problèmes pour lesquels un solution peut être trouvée en un temps polynomial (fonction de x^2, x^3, etc.)
  - des problèmes beaucoup plus complexes, de complexité exponentielle voire sub-exponentielle
- La classe P concerne les problèmes pour lesquels il existe une solution en temps polynomial
- La classe NP concerne les problème pour lesquels la vérification d'une solution se fait en temps polynomial (et non pas calcul en temps non polynomial !)
- La conjecture actuelle : $P \neq NP$ (Problème du millénaire : $http://www.claymath.org/millennium/P_vs_NP/$)

Chercher, c'est difficile ...

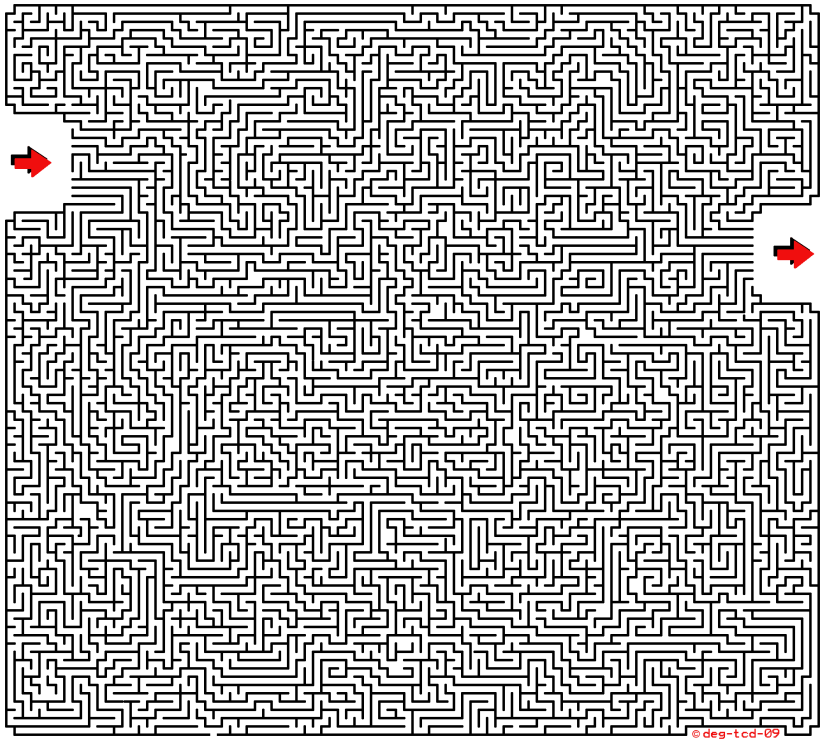


Figure: Trouver la solution ... complexe ! (source : canardpc.com)

... vérifier, c'est simple !

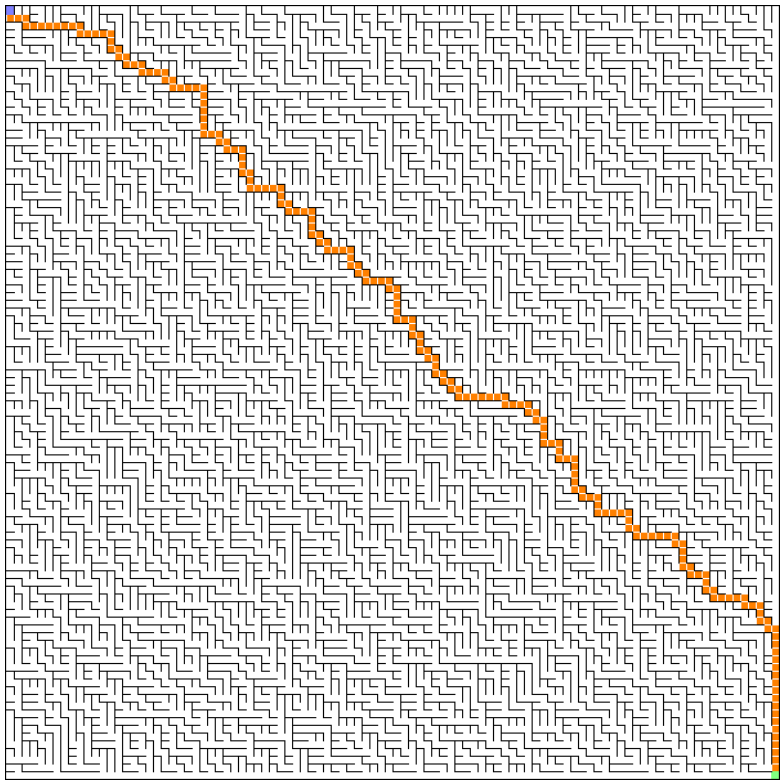


Figure: Valider la solution ... simple ! (source : Site du Zéro)

La complexité des problèmes

- Nombre d'opérations en fonction de la complexité :

N	O(ln N)	$O(N^2)$	$O(2^N)$
10	2,3	100	1024
50	3,9	2500	$1,12.10^{15}$
100	4,6	10000	$1,2.10^{30}$
1000	6,9	10^6	10^{301}

- A titre de comparaison, l'univers visible contient environ 10^{80} atomes

La complexité des problèmes : remarque sur la complexité de Kolmogorov

- Complexité d'un objet (nombre, théorie, ...) = taille maximale d'un programme calculant cet objet
- Conséquences :
  - pour générer des nombres aléatoires, il n'existe pas de programme plus court que celui qui énumère la liste des chiffres composants ces nombres
  - pour évaluer la complexité d'une théorie, il faut déterminer un programme qui la "calcule" et le comparer avec d'autres
- Cette notion est très liée au principe du "rasoir d'OCCAM" : "Pluritas non est ponenda sine necessitate"
  - "les multiples ne doivent pas être utilisés sans nécessité"
  - autrement dit, "parmi toutes les hypothèses cohérentes, il faut choisir la plus simple" !
- Globalement, un objet complexe n'a pas de description courte : pour décrire une suite aléatoire, il faut tout écrire !

La complexité des problèmes : remarque sur la complexité de Kolmogorov

- Leonid Levin (logicien) a par ailleurs utilisé la complexité de Kolmogorov pour caractériser la "régularité des structures" :
  - K(s) : complexité de Kolmogorov d'une suite s (s : suite de "0" et de "1")
  - mesure de Levin : $m(s) = 1/2^{K(s)}$
  - K(s) peut être obtenu avec des programmes de compression sans perte
  - la taille du fichier compressé de s est une valeur approchée de K(s)
- Plus un objet contient de régularités, plus il est simple au sens de Kolmogorov (K(s) faible), et plus m(s)est élevé
- "Un objet structuré est donc plus probable qu'un objet aléatoire" ...
- En conséquence : pour un algorithme, il ne suffit pas de considérer sa complexité intrinsèque, il faut la pondérer par la probabilité d'apparition des données, sachant que le pire cas est fort probable (application d'un algorithme de tri sur des données partiellement triées par exemple)

La complexité des problèmes

- En prenant comme référence l'ordinateur K Computer de RIKEN Advanced Institute for Computational Science (AICS) à Kobe - Japon
- D'une capacité de 8 PétaFlops (8.10^{15} opérations par seconde)
- Plus de 700 000 processeurs SPARC 64 cœurs
- Pour N = 100 dans le cas d'un problème d'une complexité 2^N, il faudrait environ $1,510^{14}$ secondes soit environ 4.756.488 années de calcul !

La complexité des problèmes

- Tout le problème du programmeur consiste à savoir si le problème qu'il est en train de résoudre ne se réduit pas à un problème de la classe NP
- Si tel est le cas, tout n'est pas désespéré :
  - il est parfois possible de simplifier le problème (réduction d'hypothèses, conditions d'application du calcul, algorithmes permettant un calcul de valeurs approchées, etc.)
  - il existe aussi parfois des heuristiques
- Un heuristique donne une solution approchée au problème, une "moins mauvaise solution"
- Souvent les heuristiques sont utilisées dans le cas "d'explosions combinatoires" qui se traduisent par des arbres de recherche de solution, arbres très conséquents en taille et de ce fait ingérables
- Dans ce cas, une heuristiques n'explorera pas toutes les possibilités mais certaines branches en fonction de critères précis (fonctions convergentes)

La complexité des algorithmes

- La complexité se mesure en temps, en mémoire, etc.
- Pour un algorithme, elle se mesure aussi "au minimum", "en moyenne" et "au pire"
- L'estimation du niveau de complexité d'un algorithme de détermine au cas par cas et selon les données utilisées
- Les structures de données ont une grande importance ...

La complexité des algorithmes

Input : Un tableau d'entiers non trié T : $[1..n]$
Output : Le tableau T trié

```
1 for i ← n to 1 do
2 |   for j ← 2 to i do
3 |   |   if T[j − 1] ≥ T[j] then
4 |   |   |   Permuter a[j-1] et a[j];
5 |   |   end
6 |   end
7 end
```

Algorithme 1: Tri à bulles

La complexité des algorithmes

- Pour définir la complexité du tri à bulle (comme de tout autre algorithme) :
  - il faut tout d'abord faire le choix d'une opération élémentaire
  - Cette opération doit avoir un coût fixe, c-à-d un coût identique quelque soit la valeur de l'itération
  - Dans le cas du tri à bulles, le bloc "Si $(T[j-1] \geq T[j])$ ... End" est considéré comme "élémentaire"
  - Il faut ensuite calculer le coût des boucles
  - La boucle interne, pour un i fixé, entraîne (n-(i+1)+1) soit encore (n-i) itérations pour j
  - La boucle externe revient donc à sommer les (n-i)sur i allant de 1 à n : $\sum_{i=1}^{n}(n-i)$
  - Cela donne : $\sum_{i=1}^{n}(n-i) = \frac{((n-1)-1+1)(n-1+n-(n-1))}{2} = \frac{n(n-1)}{2}$
  - La complexité (au pire et en moyenne) est en $O(n^2)$

La complexité des algorithmes

- Implémentation du tri à bulles en Java

```
static int[] table = new int[10];

static void TriBulle() {
    int n = table.length - 1;
    for (int i = n; i>= 1; i--) {
        for (int j = 2; j <= i; j++) {
            if (table[j- 1] > table[j]) {
                int temp = table[j - 1];
                table[j - 1] = table[j];
                table[j] = temp;
            }
        }
    }
}
```

La complexité des algorithmes

- Les algorithmes de tri représentent une part importante du CPU utilisé (environ 25 %) et la question relative à leur complexité est fondamentale
- Il y a d'autres domaines pour lesquels la détermination de la complexité est importante
- Exemple : calcul matriciel (taille des matrices, matrices creuses, représentation en mémoire, ordres dans les calculs, etc.)
- Pour les calculs pour lesquels un algorithme ne permet pas d'obtenir un résultat en temps raisonnable, il faut explorer la voie des heuristiques ("banch and bound", $\alpha - \beta$, A^*, etc.)
- Dans tous les cas, le temps de calcul implique une consommation d'énergie ... c'est un problème critique à la fois pour l'environnement et la consommation électrique des centres de calcul
- En conséquence : **optimisez vos codes !**