Classes préparatoires aux grandes écoles

Filière scientifique

Voies Mathématiques, physique, ingénierie et informatique (MP2I) et Mathématiques, physique, informatique (MPI)

Annexe 1

Programmes d'informatique

Programme d'informatique
Filière Mathématiques, Physique, Informatique (MPI)
Première et deuxième années

Table des matières

Introduction au programme

Les objectifs du programme L'enseignment d'informatique de classe préparatoire MPI a pour objectif la formation de futurs ingénieures et ingénieurs, enseignantes et enseignants, chercheuses et chercheurs et avant tout des personnes informées, capables de gouverner leur vie professionnelle et citoyenne nourrie par les pratiques de la démarche scientifique, en pleine connaissance et maîtrise des techniques et des enjeux de l'informatique.
Le présent programme a pour ambition de poser les bases d'un enseignement cohérent et mesuré d'une science informatique encore jeune et dont les manifestations technologiques connaissent des cycles d'obsolescence rapide. On garde donc à l'esprit :

— de privilégier la présentation de concepts fondamentaux pérennes sans s'attacher outre mesure à la description de technologies, protocoles ou normes actuels ;
— de donner aux futurs diplômées et diplômés les moyens de réussir dans un domaine en mutation rapide et dont les technologies qui en sont issues peuvent sauter brutalement d'un paradigme à un autre très différent ;
— de préparer les étudiantes et étudiants à tout un panel de professions et de situations de la vie professionnelle qui les amène à remplir tour à tour une mission d'expertise, de création ou d'invention, de prescription de méthodes ou de techniques, de contrôle critique des choix opérés ou encore de décision en interaction avec des spécialistes ;
— d'enseigner de manière à donner aux étudiantes et étudiants la flexibilité de travailler dans de nombreuses disciplines, l'informatique étant un domaine vaste qui se connecte à et tire parti de nombreuses autres disciplines.

Compétences visées Au delà de l'acquisition d'un bagage substantiel de connaissances et de méthodes de l'informatique, ce programme vise à développer les six grandes compétences suivantes :

analyser et modéliser un problème ou une situation, notamment en utilisant les objets conceptuels de l'informatique pertinents (table relationnelle, graphe, arbre, automate, modèle abstrait d'ordonnancement, etc.) ;

imaginer et concevoir une solution, décomposer en blocs, se ramener à des sous-problèmes simples et indépendants, adopter une stratégie appropriée, décrire une démarche, un algorithme ou une structure de données permettant de résoudre le problème ;

décrire et spécifier une syntaxe, les caractéristiques d'un processus, les données d'un problème, ou celles manipulées par un algorithme ou une fonction en utilisant le formalisme approprié (notamment langue française, formule logique, grammaire formelle) ;

mettre en œuvre une solution, par le choix d'un langage, par la traduction d'un algorithme ou d'une structure de données dans un langage de programmation ou un langage de requête ;

justifier et critiquer une solution, que ce soit en démontrant un algorithme par une preuve mathématique, en développant des processus d'évaluation, de contrôle, de validation d'un code que l'on a produit ou en écrivant une preuve au sein d'un système formel ;

communiquer à l'écrit ou à l'oral, présenter des travaux informatiques, une problématique et sa solution ; défendre ses choix ; documenter sa production et son implémentation.

L'enseignement de ce programme ne saurait rester aveugle aux questions sociales, juridiques, éthiques et culturelles inhérentes à la discipline de l'informatique. Ces enjeux deviennent particulièrement prégnants eu égard au rôle croissant que jouent l'intelligence artificielle et les techniques d'analyse de données dans la technologie contemporaine. La professeure ou le professeur expose ses étudiants et étudiantes à l'interaction des questions éthiques et des problèmes techniques qui jouent un rôle important dans le développement des algorithmes et des systèmes informatiques.

Sur les partis pris par le programme Ce programme impose aussi souvent que possible des choix de vocabulaire ou de notation de certaines notions. Les choix opérés ne présument pas la supériorité de l'option retenue. Ils ont été précisés dans l'unique but d'aligner les pratiques d'une classe à une autre et d'éviter l'introduction de longues définitions récapitulatives préliminaires à un exercice ou un problème. Quand des termes peu usités ont été clarifiés par leur traduction en anglais, seul le libellé en langue française est au programme. De même, ce programme nomme aussi souvent que possible l'un des algorithmes parmi les classiques qui répondent à un problème donné. Là encore, le programme ne défend pas la prééminence d'un algorithme ou d'une méthode par rapport à un autre mais il invite à faire bien plutôt que beaucoup.

Sur les langages et la programmation L'enseignement du présent programme repose sur un langage de manipulation de données (SQL) ainsi que deux langages de programmation, C et OCaml. Des annexes listent de façon limitative les éléments de ces langages qui sont exigibles des étudiants ainsi que ceux auxquels les étudiants sont familiarisés et qui peuvent être attendus à condition qu'ils soient accompagnés d'une documentation. Après des enseignements centrés sur les langages enseignés dans les classes du secondaire (au jour de l'écriture de ce programme : Scratch et Python), ces trois nouveaux langages de natures très différentes permettent d'approfondir le multilinguisme des étudiants tout en illustrant la diversité des paradigmes de programmation ou la diversité des moyens de contrôler les ressources de la machine physique et de les abstraire.
L'apprentissage du langage C conduit en particulier les étudiants à adopter immédiatement une bonne discipline de programmation tout en se concentrant sur le noyau du langage plutôt que sur une API pléthorique. En tant que langage dit de bas niveau d'abstraction utilisé entre autres pour écrire tous les systèmes d'exploitation, il permet une gestion explicite de la mémoire et des ressources de la machine, indispensable dans le cas où celles-ci sont limitées (systèmes embarqués, mobiles).
L'apprentissage du langage OCaml permet en particulier aux étudiants de recourir rapidement à un niveau d'abstraction supérieur et de manipuler facilement des structures de données récursives. Pour autant, son utilisation peut également simplifier certaines manipulations sur les fils d'exécution *(threads)*, par exemple. La plupart des algorithmes qui figurent au programme se prêtent indifféremment à une programmation en C ou en OCaml. On veille à développer de façon parallèle les compétences de programmation dans ces deux langages.
Il convient de ne pas axer uniquement l'enseignement de ce programme sur le développement de compétences en programmation : si la capacité à écrire des programmes courts, précis, agréables à lire et documentés fait partie d'une formation exhaustive en informatique, un accent trop important sur l'écriture de code peut donner une vision étroite et trompeuse de la place de la programmation dans la discipline informatique. Les défauts, les bogues et les failles de logique constituent systématiquement la cause première des vulnérabilités des logiciels exploitées de façon malveillante. La vigilance vis-à-vis de pratiques de programmation sûres est apprise dès les premiers stades de l'apprentissage de la programmation. On s'attache à sensibiliser les étudiants à ces techniques, à la prévention des vulnérabilités et à une validation formelle ou expérimentale rigoureuse des résultats obtenus. Les étudiants sont incités à analyser les sources possibles d'invalidité des données manipulées par leurs programmes, y compris en cas d'exécution concurrente, et à savoir appliquer des principes de programmation défensive.

Mode d'emploi Pour une meilleure lisibilité de l'ensemble, les acquis d'apprentissage finaux ont été structurés par chapitres thématiques, sans chercher à éviter une redondance qui ne fait que témoigner des liens que ces thèmes entretiennent. Des repères temporels peuvent être proposés mais l'organisation de la progression au sein de ces acquis relève de la responsabilité pédagogique de la professeure ou du professeur et le tissage de liens entre les thèmes contribue à la valeur de son enseignement. Les symboles (S1), (S2) et (S3-4) indiquent que les notions associées sont étudiées avant la fin du premier ou du second semestre de la première année, ou durant la deuxième année, respectivement ; ces notions sont régulièrement revisitées tout au long des deux années d'enseignement. Lorsqu'une telle spécification s'applique uniformément à une section ou sous-section, elle n'est pas répétée aux niveaux inférieurs.

1 Méthodes de programmation S1 S2 S3-4

Le programme construit une progression à partir des acquis du lycée en matière d'algorithmique et programmation, dont on rappelle qu'ils ont permis, *a minima*, de rencontrer les notions de variables, de type, d'affectation, d'instruction conditionnelle, de boucles conditionnelles ou inconditionnelles et de manipuler de façon simple les listes en Python.

1.1 Algorithmes et programmes S1

Ce paragraphe introduit notamment le principe de validation d'un algorithme ou d'un programme et celui d'étude de son efficacité, qui sont pratiqués tout au long des deux années.

Notions	Commentaires
Notion de programme comme mise en œuvre d'un algorithme. Paradigme impératif structuré, paradigme déclaratif fonctionnel, paradigme logique.	On ne présente pas de théorie générale sur les paradigmes de programmation, on se contente d'observer les paradigmes employés sur des exemples. La notion de saut inconditionnel (instruction `GOTO`) est hors programme. On mentionne le paradigme logique uniquement à l'occasion de la présentation des bases de données.
Caractère compilé ou interprété d'un langage.	Transformation d'un fichier texte source en un fichier objet puis en un fichier exécutable. Différence entre fichiers d'interface et fichiers d'implémentation.
Représentation des flottants. Problèmes de précision des calculs flottants.	On illustre l'impact de la représentation par des exemples de divergence entre le calcul théorique d'un algorithme et les valeurs calculées par un programme. Les comparaisons entre flottants prennent en compte la précision.
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.
Analyse de la complexité d'un algorithme. Complexité dans le pire cas, dans le cas moyen. Notion de coût amorti.	On limite l'étude de la complexité dans le cas moyen et du coût amorti à quelques exemples simples.

1.2 Discipline de programmation S1 S2 S3-4

Ce paragraphe définit une discipline de programmation qui a vocation à être observée dès le début et durant toute la durée des deux années d'enseignement.

Notions	Commentaires
Spécification des données attendues en entrée, et fournies en sortie/retour.	On entraîne les étudiants à accompagner leurs programmes et leurs fonctions d'une spécification. Les signatures des fonctions sont toujours précisées.
Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante.	Ces annotations se font à l'aide de commentaires.
Programmation défensive. Assertion. Sortie du programme ou exception levée en cas d'évaluation négative d'une assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées ou pour le contrôle de débordements. Plus généralement, les étudiants sont sensibilisés à réfléchir aux causes possibles (internes ou externes à leur programme) d'opérer sur des données invalides et à adopter un style de programmation défensif. Les étudiants sont sensibilisés à la différence de garanties apportées selon les langages, avec l'exemple d'un typage faible en C et fort en OCaml. On veille à ne pas laisser penser que les exceptions servent uniquement à gérer des erreurs.
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase.

1.3 Validation, test (S1)

La validation de code par sa soumission à des jeux de tests est une phase essentielle du cycle de développement logiciel. On en fait percevoir l'importance dès le début et durant toute la durée des deux années d'enseignement.

Notions	**Commentaires**
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant est capable d'écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.
Graphe de flot de contrôle. Chemins faisables. Couverture des sommets, des arcs ou des chemins (avec ou sans cycle) du graphe de flot de contrôle.	Les étudiants sont capables d'écrire un jeu de tests satisfaisant un critère de couverture des instructions (sommets) ou des branches (arcs) sur les chemins faisables.
Test exhaustif de la condition d'une boucle ou d'une conditionnelle.	Il s'agit, lorsque la condition booléenne comporte des conjonctions ou disjonctions, de ne pas se contenter de la traiter comme étant globalement vraie ou fausse mais de formuler des tests qui réalisent toutes les possibilités de la satisfaire. On se limite à des exemples simples pour lesquels les cas possibles se décèlent dès la lecture du programme.

2 Récursivité et induction S1 S2

La capacité d'un programme à faire appel à lui-même est un concept primordial en informatique. Historiquement, l'auto-référence est au cœur du paradigme de programmation fonctionnelle. Elle imprègne aujourd'hui, de manière plus ou moins marquée, la plupart des langages de programmation contemporains. Le principe d'induction est une notion fondamentale et transverse à l'ensemble de ce programme. Il permet d'écrire des démonstrations avec facilité dès que l'on s'intéresse à toute sorte de structures (arbres, formules de logiques, classes de langage, etc.).

Notions	**Commentaires**
Récursivité d'une fonction. Récursivité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples. S1	On se limite à une présentation pratique de la récursivité comme technique de programmation. Les récurrences usuelles : $T(n) = T(n-1) + an$, $T(n) = aT(n/2) + b$, ou $T(n) = 2T(n/2) + f(n)$ sont introduites au fur et à mesure de l'étude de la complexité des différents algorithmes rencontrés. On utilise des encadrements élémentaires *ad hoc* afin de les justifier ; on évite d'appliquer un théorème-maître général.
Ensemble ordonné, prédécesseur et successeur, prédécesseur et successeur immédiat. Élément minimal. Ordre produit, ordre lexicographique. Ordre bien fondé. S2	On fait le lien avec la notion d'accessibilité dans un graphe orienté acyclique. L'objectif n'est pas d'étudier la théorie abstraite des ensembles ordonnés mais de poser les définitions et la terminologie.
Ensemble inductif, défini comme le plus petit ensemble engendré par un système d'assertions et de règles d'inférence. Ordre induit. Preuve par induction structurelle. S2	On insiste sur les aspects pratiques : construction de structure de données et filtrage par motif. On présente la preuve par induction comme une généralisation de la preuve par récurrence.

Mise en œuvre
On met l'accent sur la gestion au niveau de la machine, en termes d'occupation mémoire, de la pile d'exécution, et de temps de calcul, en évoquant les questions de sauvegarde et de restauration de contexte. On évite de se limiter à des exemples informatiquement peu pertinents (factorielle, suite de Fibonacci, …). Toute théorie générale de la dérécursification est hors programme. Un étudiant peut mener des raisonnements par induction structurelle.

3 Structures de données S1 S2 S3-4

On insiste sur le fait que le développement d'un algorithme va de pair avec la conception d'une structure de données taillée à la mesure du problème que l'on cherche à résoudre et des opérations sur les données que l'on est amené à répéter.

3.1 Types et abstraction S1

Notions	Commentaires
Type prédéfini (booléen, entier, flottant). Pointeur. Type paramétré (tableau). Type composé. Tableaux statiques. Allocation (`malloc`) et désallocation (`free`) dynamique.	On se limite à une présentation pratique des types, en les illustrant avec les langages du programme. Un étudiant est capable d'inférer un type à la lecture d'un fragment de code, cependant toute théorie du typage est hors programme.
Définition d'une structure de données abstraite comme un type muni d'opérations.	On parle de constructeur pour l'initialisation d'une structure, d'accesseur pour récupérer une valeur et de transformateur pour modifier l'état de la structure. On montre l'intérêt d'une structure de données abstraite en terme de modularité. On distingue la notion de structure de données abstraite de son implémentation. Plusieurs implémentations concrètes sont interchangeables. La notion de classe et la programmation orientée objet sont hors programme.
Distinction entre structure de données mutable et immuable.	Illustrée en langage OCaml.

Mise en œuvre
Il s'agit de montrer l'intérêt et l'influence des structures de données sur les algorithmes et les méthodes de programmation. On insiste sur la distinction entre une structure de données abstraite (un type muni d'opérations ou encore une interface) et son implémentation concrète. On montre l'intérêt d'une structure de données abstraite en terme de modularité. Grâce aux bibliothèques, on peut utiliser des structures de données avant d'avoir programmé leur réalisation concrète.

3.2 Structures de données séquentielles S1 S2

Notions	Commentaires
Structure de liste. Implémentation par un tableau, par des maillons chaînés. S1	On insiste sur le coût des opérations selon le choix de l'implémentation. Pour l'implémentation par un tableau, on se fixe une taille maximale. On peut évoquer le problème du redimensionnement d'un tableau.
Structure de pile. Structure de file. Implémentation par un tableau, par des maillons chaînés. S1	
Structure de tableau associatif implémenté par une table de hachage. S2	La construction d'une fonction de hachage et les méthodes de gestion des collisions éventuelles ne sont pas des exigibles du programme.
Sérialisation. S2	On présente un exemple de sérialisation d'une structure hiérarchique et d'une structure relationnelle.

Mise en œuvre
On présente les structures de données construites à l'aide de pointeurs d'abord au tableau avant de guider les étudiants dans l'implémentation d'une telle structure.

3.3 Structures de données hiérarchiques (S2)(S3-4)

Notions	Commentaires
Définition inductive du type arbre binaire. Vocabulaire : nœud, nœud interne, racine, feuille, fils, père, hauteur d'un arbre, profondeur d'un nœud, étiquette, sous-arbre. (S2)	La hauteur de l'arbre vide est -1. On mentionne la représentation d'un arbre complet dans un tableau.
Arbre. Conversion d'un arbre d'arité quelconque en un arbre binaire. (S2)	La présentation donne lieu à des illustrations au choix du professeur. Il peut s'agir par exemple d'expressions arithmétiques, d'arbres préfixes (*trie*), d'arbres de décision, de dendrogrammes, d'arbres de classification, etc.
Parcours d'arbre. Ordre préfixe, infixe et postfixe. (S2)	On peut évoquer le lien avec l'empilement de blocs d'activation lors de l'appel à une fonction récursive.
Implémentation d'un tableau associatif par un arbre binaire de recherche. Arbre bicolore. (S2)	On note l'importance de munir l'ensemble des clés d'un ordre total.
Propriété de tas. Structure de file de priorité implémentée par un arbre binaire ayant la propriété de tas. (S2)	Tri par tas.
Structure unir & trouver pour la représentation des classes d'équivalence d'un ensemble. Implémentation par des arbres. (S3-4)	On commence par donner des implémentations naïves de la structure unir & trouver qui privilégient soit l'opération unir, soit l'opération trouver, avant de donner une implémentation par des arbres qui permet une mise en œuvre efficace des deux opérations. L'analyse de la complexité de cette structure est admise.

Mise en œuvre
On présente les manipulations usuelles sur les arbres en C et en OCaml. Il n'est pas attendu d'un étudiant une maîtrise technique de l'écriture du code d'une structure de données arborescente mutable à l'aide de pointeurs, mais il est attendu qu'il sache l'utiliser.

3.4 Structures de données relationnelles (S2)

Il s'agit de définir le modèle des graphes, leurs représentations et leurs manipulations.
On s'efforce de mettre en avant des applications importantes et si possibles modernes : réseau de transport, graphe du web, réseaux sociaux, bio-informatique. On précise autant que possible la taille typique de tels graphes.

Notions	Commentaires
Graphe orienté, graphe non orienté. Sommet (ou nœud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité, forte connexité. Graphe orienté acyclique. Arbre en tant que graphe connexe acyclique. Forêt. Graphe biparti.	Notation : graphe $G = (S, A)$, degrés $d_+(s)$ et $d_-(s)$ dans le cas orienté. On n'évoque pas les multi-arcs. On représente un graphe orienté par une matrice d'adjacence ou par des listes d'adjacence.
Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe.	On motive l'ajout d'information à un graphe par des exemples concrets : graphe de distance, automate fini, diagramme de décision binaire.

Mise en œuvre
On présente les manipulations usuelles sur les graphes en C et en OCaml. La présentation en C s'effectue à travers des tableaux statiques. Pour la représentation en liste d'adjacence, on peut considérer un tableau à deux dimensions dont les lignes représentent chaque liste avec une sentinelle ou un indicateur de taille en premier indice.

4 Algorithmique S2 S3-4

Les algorithmes sont présentés au tableau en spécifiant systématiquement les entrées et sorties et en étudiant, dans la mesure du possible, leur correction et leur complexité.

4.1 Algorithmes probabilistes, algorithmes d'approximation S3-4

Notions	Commentaires
Algorithme déterministe. Algorithme probabiliste (*Las Vegas* et *Monte Carlo*).	On s'en tient aux définitions et à des exemples choisis par le professeur. On mentionne l'intérêt d'une méthode *Las Vegas* pour construire un objet difficile à produire par une méthode déterministe (par exemple, construction d'un nombre premier de taille cryptographique). Quelques exemples possibles : k-ième minimum d'un tableau non trié, problème des huit reines, etc.
Problème de décision. Problème d'optimisation. Instance d'un problème, fonction de coût. Notion d'algorithme d'approximation.	Seule la notion d'algorithme d'approximation est au programme. L'étude de techniques générales d'approximation est hors programme. On indique, par exemple sur le problème MAX2SAT, que la méthode probabiliste peut fournir de bons algorithmes d'approximation.

4.2 Exploration exhaustive S2 S3-4

Notions	Commentaires
Recherche par force brute. Retour sur trace (*Backtracking*). S2	On peut évoquer l'intérêt d'ordonner les données avant de les parcourir (par exemple par une droite de balayage).
Algorithme par séparation et évaluation (*Branch and bound*). S3-4	On peut évoquer sur des exemples quelques techniques d'évaluation comme les méthodes de relaxation (par exemple la relaxation continue).
Mise en œuvre	
L'objectif est de donner des outils de conception d'algorithmes et de parvenir à ce que les étudiants puissent, dans une situation simple, sélectionner une stratégie pertinente par eux-mêmes et la mettre en œuvre de façon autonome. Dans les cas les plus complexes, les choix et les recommandations d'implémentation sont guidés.	

4.3 Décomposition d'un problème en sous-problèmes S2 S3-4

Notions	Commentaires
Algorithme glouton fournissant une solution exacte. S2	On peut traiter comme exemples d'algorithmes exacts : codage de Huffman, sélection d'activité, ordonnancement de tâches unitaires avec pénalités de retard sur une machine unique.
Exemple d'algorithme d'approximation fourni par la méthode gloutonne. S3-4	On peut traiter par exemple : couverture des sommets dans un graphe, problème du sac à dos en ordonnant les objets.
Diviser pour régner. Rencontre au milieu. Dichotomie. S2	On peut traiter un ou plusieurs exemples comme : tri par partition-fusion, comptage du nombre d'inversions dans une liste, calcul des deux points les plus proches dans une ensemble de points ; recherche d'un sous-ensemble d'un ensemble d'entiers dont la somme des éléments est donnée ; recherche dichotomique dans un tableau trié. On présente un exemple de dichotomie où son recours n'est pas évident : par exemple, la couverture de n points de la droite par k segments égaux de plus petite longueur.

Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes. Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée. (S2)	On souligne les enjeux de complexité en mémoire. On peut traiter un ou plusieurs exemples comme : problème de la somme d'un sous-ensemble, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein).

Mise en œuvre
L'objectif est de donner des outils de conception d'algorithmes et de parvenir à ce que les étudiants puissent, dans une situation simple, sélectionner une stratégie pertinente par eux-mêmes et la mettre en œuvre de façon autonome. Dans les cas les plus complexes, les choix et les recommandations d'implémentation sont guidés. Les listes d'exemples cités en commentaires ne sont ni impératives ni limitatives.

4.4 Algorithmique des textes (S2)

Notions	**Commentaires**
Recherche dans un texte. Algorithme de Boyer-Moore. Algorithme de Rabin-Karp.	On peut se restreindre à une version simplifiée de l'algorithme de Boyer-Moore, avec une seule fonction de décalage. L'étude précise de la complexité de ces algorithmes n'est pas exigible.
Compression. Algorithme de Huffman. Algorithme Lempel-Ziv-Welch.	On explicite les méthodes de décompression associées.

4.5 Algorithmique des graphes (S2)(S3-4)

Notions	**Commentaires**
Notion de parcours (sans contrainte). Notion de parcours en largeur, en profondeur. Notion d'arborescence d'un parcours. (S2)	On peut évoquer la recherche de cycle, la bicolorabilité d'un graphe, la recherche de plus courts chemins dans un graphe à distance unitaire.
Accessibilité. Tri topologique d'un graphe orienté acyclique à partir de parcours en profondeur. Recherche des composantes connexes d'un graphe non orienté. (S2)	On fait le lien entre accessibilité dans un graphe orienté acyclique et ordre.
Recherche des composantes fortement connexes d'un graphe orienté par l'algorithme de Kosaraju. (S3-4)	On fait le lien entre composantes fortement connexes et le problème 2-SAT.
Notion de plus courts chemins dans un graphe pondéré. Algorithme de Dijkstra. Algorithme de Floyd-Warshall. (S2)	On présente l'algorithme de Dijkstra avec une file de priorité en lien avec la représentation de graphes par listes d'adjacences. On présente l'algorithme de Floyd-Warshall en lien avec la représentation de graphes par matrice d'adjacence.
Recherche d'un arbre couvrant de poids minimum par l'algorithme de Kruskal. (S3-4)	On peut mentionner l'adaptation au problème du chemin le plus large dans un graphe non-orienté.
Recherche d'un couplage de cardinal maximum dans un graphe biparti par des chemins augmentants. (S3-4)	On se limite à une approche élémentaire; l'algorithme de Hopcroft-Karp n'est pas au programme. Les graphes bipartis et couplages sont introduits comme outils naturels de modélisation; ils peuvent également constituer une introduction aux problèmes de flots.

Mise en œuvre
Une attention particulière est portée sur le choix judicieux du mode de représentation d'un graphe en fonction de l'application et du problème considéré. On étudie en conséquence l'impact de la représentation sur la conception d'un algorithme et sur sa complexité (en temps et en espace). On se concentre sur l'approfondissement des algorithmes cités dans le programme et le ré-emploi de leurs idées afin de résoudre des problèmes similaires. La connaissance d'une bibliothèque d'algorithmes fonctionnant sur des principes différents mais résolvant un même problème n'est pas un objectif du programme.

4.6 Algorithmique pour l'intelligence artificielle et l'étude des jeux S3-4

Cette partie permet d'introduire les concepts d'apprentissage, de stratégie et d'heuristique. Ce dernier est abordé par des exemples où l'heuristique est précisément définie mais sans en évaluer la performance.

Notions	**Commentaires**
Apprentissage supervisé.	Algorithme des k plus proches voisins avec distance euclidienne. Arbres k dimensionnels. Apprentissage d'arbre de décision : algorithme ID3 restreint au cas d'arbres binaires. Matrice de confusion. On observe des situations de surapprentissage sur des exemples.
Apprentissage non-supervisé.	Algorithme de classification hiérarchique ascendante. Algorithme des k-moyennes. La démonstration de la convergence n'est pas au programme. On observe des convergences vers des minima locaux.
Jeux d'accessibilité à deux joueurs sur un graphe. Stratégie. Stratégie gagnante. Position gagnante. Détermination des positions gagnantes par le calcul des attracteurs. Construction de stratégies gagnantes.	On considère des jeux à deux joueurs (J_1 et J_2) modélisés par des graphes bipartis (l'ensemble des états contrôlés par J_1 et l'ensemble des états contrôlés par J_2). Il y a trois types d'états finals : les états gagnants pour J_1, les états gagnants pour J_2 et les états de match nul. On ne considère que les stratégies sans mémoire.
Notion d'heuristique. Algorithme min-max avec une heuristique. Élagage alpha-beta.	
Graphe d'états. Recherche informée : algorithme A*.	On souligne l'importance de l'admissibilité de l'heuristique, ainsi que le cas où l'heuristique est également monotone.

Mise en œuvre
La connaissance des théories sous-jacentes aux algorithmes de cette section n'est pas un attendu du programme. Les étudiants acquièrent une familiarité avec les idées qu'ils peuvent réinvestir dans des situations où les modélisations et les recommandations d'implémentation sont guidées.

5 Gestion des ressources de la machine S1 S3-4

Le programme vise à donner un premier aperçu des liens qu'assurent les systèmes d'exploitation (plus largement les plates-formes d'exécution) entre les programmes et les ressources offertes par les machines qui les exécutent. Le fonctionnement du matériel, l'architecture des ordinateurs, la conception des systèmes, la gestion des interfaces, les protocoles de communication, la virtualisation (de la mémoire, des processeurs, etc.) sont hors programme. Ce programme se focalise sur trois aspects de la gestion de la machine :

— la mémoire au sein d'un processus qui exécute un programme ;
— les systèmes de fichiers qui permettent d'interagir avec un processus, en entrée et sortie ;
— la concurrence au sein des processus par des fils d'exécution, exploitant les possibilités d'exécution concurrente des processeurs actuels.

Bien que ces notions soient indépendantes du système d'exploitation, le système Linux est le plus propice pour introduire les éléments de ce programme.

5.1 Gestion de la mémoire d'un programme S1

Notions	Commentaires
Utilisation de la pile et du tas par un programme compilé.	On présente l'allocation des variables globales, le bloc d'activation d'un appel.
Notion de portée syntaxique et durée de vie d'une variable. Allocation des variables locales et paramètres sur la pile.	On indique la répartition selon la nature des variables : globales, locales, paramètres.
Allocation dynamique.	On présente les notions en lien avec le langage C : `malloc` et `free`, pointeur nul, type `void*`, transtypage, relation avec les tableaux, protection mémoire (*segmentation violation*).

5.2 Gestion des fichiers et entrées-sorties S1

Notions	Commentaires
Interface de fichiers : taille, accès séquentiel.	
Implémentation interne : blocs et nœuds d'index (*inode*).	On présente le partage de blocs (avec liens physiques ou symboliques) et l'organisation hiérarchique de l'espace de nommage.
Accès, droits et attributs.	On utilise sur des exemples les fonctions d'accès et d'écriture dans les différents modes.
Fichiers spéciaux : flux standard (entrée standard `stdin`, sortie standard `stdout`, sortie d'erreur standard `stderr`) et redirections dans l'interface système (*shell*).	On présente la notion de tube (*pipe*).

Mise en œuvre
Les seules notions exigibles sont celles permettant à un programme de gérer l'ouverture, la fermeture et l'accès à un ou plusieurs fichiers, selon les modalités précisées en annexes. On attend toutefois d'un étudiant une expérience du montage d'un support de fichiers amovible, de la gestion des droits d'accès à des parties de l'arborescence, de la création et du déplacement des parties de l'aborescence et de la gestion des liens physiques et symboliques. Le professeur expose également ses étudiants à la réalisation d'enchaînements de programmes via des tubes (*pipes*).

5.3 Gestion de la concurrence et synchronisation (S3-4)

L'apprentissage des notions liées au parallélisme d'exécution se limite au cas de fils d'exécutions (*threads*) internes à un processus, sur une machine. Les problèmes d'algorithmes répartis et les notions liées aux réseaux et à la communication asynchrone sont hors programme.

Notions	**Commentaires**
Notion de fils d'exécution. Non-déterminisme de l'exécution.	Les notions sont présentées au tableau en privilégiant le pseudo-code ; elles sont mises en œuvre au cours de travaux pratiques en utilisant les bibliothèques POSIX `pthread` (en langage C) ou `Thread` (en langage OCaml), au choix du professeur, selon les modalités précisées en annexe. On s'en tient aux notions de base : création, attente de terminaison.
Synchronisation de fils d'exécution. Algorithme de Peterson pour deux fils d'exécution. Algorithme de la boulangerie de Lamport pour plusieurs fils d'exécution.	On illustre l'importance de l'atomicité par quelques exemples et les dangers d'accès à une variable en l'absence de synchronisation. On présente les notions de mutex et sémaphores.
Mise en œuvre	
Les concepts sont illustrés sur des schémas de synchronisation classiques : rendez-vous, producteur-consommateur. Les étudiants sont également sensibilisés au non-déterminisme et aux problèmes d'interblocage et d'équité d'accès, illustrables sur le problème classique du dîner des philosophes.	

6 Logique S2 S3-4

6.1 Syntaxe des formules logiques S2

Le but de cette partie est de familiariser progressivement les étudiants avec la différence entre syntaxe et sémantique d'une part et de donner le vocabulaire permettant de modéliser une grande variété de situations (par exemple, satisfaction de contraintes, planification, diagnostic, vérification de modèles, etc.).
L'étude des quantificateurs est l'occasion de formaliser les notions de variables libres et liées, et de portée, notions que l'on retrouve dans la pratique de la programmation.

Notions	**Commentaires**
Variables propositionnelles, connecteurs logiques, arité. Formules propositionnelles, définition par induction, représentation comme un arbre. Sous-formule. Taille et hauteur d'une formule.	Notations : $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$. Les formules sont des données informatiques. On fait le lien entre les écritures d'une formule comme mot et les parcours d'arbres.
Quantificateurs universel et existentiel. Variables liées, variables libres, portée. Substitution d'une variable.	On ne soulève aucune difficulté technique sur la substitution. L'unification est hors programme.

Mise en œuvre
On implémente uniquement les formules propositionnelles sous forme d'arbres.

6.2 Sémantique de vérité du calcul propositionnel S2

Par souci d'éviter trop de technicité, on ne présente la notion de valeur de vérité que pour des formules sans quantificateurs.

Notions	**Commentaires**
Valuations, valeurs de vérité d'une formule propositionnelle. Satisfiabilité, modèle, ensemble de modèles, tautologie, antilogie. Équivalence sur les formules. Conséquence logique entre deux formules.	Notations V pour la valeur vraie, F pour la valeur fausse. Une formule est satisfiable si elle admet un modèle, tautologique si toute valuation en est un modèle. On peut être amené à ajouter à la syntaxe une formule tautologique et une formule antilogique ; elles sont en ce cas notées $\top$ et $\bot$. On présente les lois de De Morgan, le tiers exclu et la décomposition de l'implication. On étend la notion à celle de conséquence ϕ d'un ensemble de formules Γ : on note $\Gamma \models \phi$. La compacité est hors programme.
Forme normale conjonctive, forme normale disjonctive. Mise sous forme normale.	Lien entre forme normale disjonctive complète et table de vérité. On peut représenter les formes normales comme des listes de listes de littéraux. Exemple de formule dont la taille des formes normales est exponentiellement plus grande.
Problème SAT, n-SAT, algorithme de Quine.	On incarne SAT par la modélisation d'un problème (par exemple la coloration des sommets d'un graphe).

6.3 Déduction naturelle S3-4

Il s'agit de présenter les preuves comme permettant de pallier deux problèmes de la présentation précédente du calcul propositionnel : nature exponentielle de la vérification d'une tautologie, faible lien avec les preuves mathématiques.
Il ne s'agit, en revanche, que d'introduire la notion d'arbre de preuve. La déduction naturelle est présentée comme un jeu de règles d'inférence simple permettant de faire un calcul plus efficace que l'étude de la table de vérité. Toute technicité dans les preuves dans ce système est à proscrire.

Notions	**Commentaires**
Règle d'inférence, dérivation. Définition inductive d'un arbre de preuve.	Notation $\vdash$. Séquent $H_1, \dots, H_n \vdash C$. On présente des exemples tels que le *modus ponens* $(p, p \to q \vdash q)$ ou le syllogisme *barbara* $(p \to q, q \to r \vdash p \to r)$. On présente des exemples utilisant les règles précédentes.
Règles d'introduction et d'élimination de la déduction naturelle pour les formules propositionnelles. Correction de la déduction naturelle pour les formules propositionnelles.	On présente les règles pour $\wedge, \vee, \neg$ et $\to$. On écrit de petits exemples d'arbre de preuves (par exemple $\vdash (p \to q) \to \neg(p \wedge \neg q)$, etc.).
Règles d'introduction et d'élimination pour les quantificateurs universels et existentiels.	On motive ces règles par une approche sémantique intuitive.

Mise en œuvre
Il ne s'agit pas d'implémenter ces règles mais plutôt d'être capable d'écrire de petites preuves dans ce système. On peut également présenter d'autres utilisations de règles d'inférences pour raisonner.

7 Bases de données S2

On se limite volontairement à une description applicative des bases de données en langage SQL. Il s'agit de permettre d'interroger une base présentant des données à travers plusieurs relations. On ne présente ni l'algèbre relationnelle ni le calcul relationnel.

Notions	**Commentaires**
Vocabulaire des bases de données : tables ou relations, attributs ou colonnes, domaine, schéma de tables, enregistrements ou lignes, types de données.	On présente ces concepts à travers de nombreux exemples. On s'en tient à une notion sommaire de domaine : entier, flottant, chaîne ; aucune considération quant aux types des moteurs SQL n'est au programme. Aucune notion relative à la représentation des dates n'est au programme ; en tant que de besoin on s'appuie sur des types numériques ou chaîne pour lesquels la relation d'ordre coïncide avec l'écoulement du temps. Toute notion relative aux collations est hors programme ; en tant que de besoin on se place dans l'hypothèse que la relation d'ordre correspond à l'ordre lexicographique usuel.
Clé primaire.	Une clé primaire n'est pas forcément associée à un unique attribut même si c'est le cas le plus fréquent. La notion d'index est hors programme.
Entités et associations, clé étrangère.	On s'intéresse au modèle entité–association au travers de cas concrets d'associations $1-1, 1-*, *-*$. Séparation d'une association $*-*$ en deux associations $1-*$. L'utilisation de clés primaires et de clés étrangères permet de traduire en SQL les associations $1-1$ et $1-*$.
Requêtes `SELECT` avec simple clause `WHERE` (sélection), projection, renommage `AS`. Utilisation des mots-clés `DISTINCT`, `LIMIT`, `OFFSET`, `ORDER BY`. Opérateurs ensemblistes `UNION`, `INTERSECT` et `EXCEPT`, produit cartésien.	Les opérateurs au programme sont `+`, `-`, `*`, `/` (on passe outre les subtilités liées à la division entière ou flottante), `=`, `<>`, `<`, `<=`, `>`, `>=`, `AND`, `OR`, `NOT`, `IS NULL`, `IS NOT NULL`.
Jointures internes T_1 `JOIN` T_2 … `JOIN` T_n `ON` ϕ, externes à gauche T_1 `LEFT JOIN` T_2 `ON` ϕ.	On présente les jointures (internes) en lien avec la notion d'associations entre entités.
Agrégation avec les fonctions `MIN`, `MAX`, `SUM`, `AVG` et `COUNT`, y compris avec `GROUP BY`.	Pour la mise en œuvre des agrégats, on s'en tient à la norme SQL99. On présente quelques exemples de requêtes imbriquées.
Filtrage des agrégats avec `HAVING`.	On marque la différence entre `WHERE` et `HAVING` sur des exemples.

Mise en œuvre
La création, la suppression et la modification de tables au travers du langage SQL sont hors programme. La mise en œuvre effective se fait au travers d'un logiciel permettant d'interroger une base de données à l'aide de requêtes SQL. Récupérer le résultat d'une requête à partir d'un programme n'est pas un objectif. Même si aucun formalisme graphique précis n'est au programme, on peut décrire les entités et les associations qui les lient au travers de diagrammes sagittaux informels. Sont hors programme : la notion de modèle logique *vs* physique, les bases de données non relationnelles, les méthodes de modélisation de base, les fragments DDL, TCL et ACL du langage SQL, l'optimisation de requêtes par l'algèbre relationnelle.

8 Langages formels S3-4

8.1 Langages réguliers

On introduit les expressions régulières comme formalisme dénotationnel pour spécifier un motif dans le cadre d'une recherche textuelle.

Notions	Commentaires
Alphabet, mot, préfixe, suffixe, facteur, sous-mot.	Le mot vide est noté ε.
Langage comme ensemble de mots sur un alphabet. Opérations régulières sur les langages (union, concaténation, étoile de Kleene). Définition inductive des langages réguliers.	
Expression régulière. Dénotation d'un langage régulier.	On introduit les expressions régulières comme un formalisme dénotationnel pour les motifs. On note l'expression dénotant le langage vide $\emptyset$, celle dénotant le langage réduit au mot vide ε, l'union par $\vert$, la concaténation par juxtaposition et l'étoile de Kleene par une étoile.
Expressions régulières étendues.	Le lien est fait avec les expressions régulières de la norme POSIX, mais on ne développe aucune théorie supplémentaire à leur sujet et aucune connaissance au sujet de cette norme n'est exigible.

8.2 Automates finis

Les automates constituent un modèle de calcul puissant qui irrigue de nombreuses branches de l'informatique. On voit ici les automates comme un formalisme opérationnel efficace pour la recherche de motifs. On vérifie que le formalisme des automates coïncide exactement avec l'expressivité des expressions régulières.

Notions	Commentaires
Automate fini déterministe. État accessible, co-accessible. Automate émondé. Langage reconnu par un automate.	On insiste sur la richesse de systèmes dont le fonctionnement peut être modélisé par un automate.
Transition spontanée (ou ε-transition). Automate fini non déterministe.	
Déterminisation d'un automate non déterministe.	On fait le lien entre l'élimination des transitions spontanées et l'accessibilité dans un graphe. On aborde l'élimination des transitions spontanées et plus généralement les constructions d'automates à la Thompson sur des exemples, sans chercher à formaliser complètement les algorithmes sous-jacents.
Construction de l'automate de Glushkov associé à une expression régulière par l'algorithme de Berry-Sethi.	Les notions de langage local et d'expression régulière linéaire sont introduites dans cette seule perspective.
Passage d'un automate à une expression régulière. Élimination des états. Théorème de Kleene.	On se limite à la description du procédé d'élimination et à sa mise en œuvre sur des exemples d'automates de petite taille ; cela constitue la preuve du sens réciproque du théorème de Kleene.
Stabilité de la classe des langages reconnaissables par union finie, intersection finie, complémentaire.	
Lemme de l'étoile.	Soit L le langage reconnu par un automate à n états : pour tout $u \in L$ tel que $\vert u\vert \geq n$, il existe x, y, z tels que $u = xyz$, $\vert xy\vert \leq n$, $y \neq \varepsilon$ et $xy^*z \subseteq L$.

8.3 Grammaires non contextuelles

Les grammaires formelles ont pour principal intérêt de définir des syntaxes structurées, en particulier celles des langages informatiques (langage de programmation, langage de requête, langage de balisage, etc.). On s'intéresse surtout à la manière dont les mots s'obtiennent par la grammaire et, de façon modeste, à la manière d'analyser un mot (un programme) en une structure de données qui le représente.

Notions	**Commentaires**
Grammaire non contextuelle. Vocabulaire : symbole initial, symbole non-terminal, symbole terminal, règle de production, dérivation immédiate, dérivation. Langage engendré par une grammaire, langage non contextuel. Non contextualité des langages réguliers.	Notations : règle de production $\rightarrow$, dérivation immédiate $\Rightarrow$, dérivation $\Rightarrow^*$. On montre comment définir une expression arithmétique ou une formule de la logique propositionnelle par une grammaire. On peut présenter comme exemple un mini-langage fictif de programmation ou un mini-langage de balisage. Sont hors programme : les automates à pile, les grammaires syntagmatiques générales, la hiérarchie de Chomsky.
Arbre d'analyse. Dérivation à gauche, à droite. Ambiguïté d'une grammaire. Équivalence faible.	On présente le problème du « sinon pendant » (*dangling else*).
Exemple d'algorithme d'analyse syntaxique.	On peut présenter au tableau un algorithme *ad hoc* d'analyse syntaxique par descente récursive (algorithme *top-down*) pour un langage de balisage fictif (par exemple, la grammaire de symbole initial S et de règles de production $S \rightarrow TS|c, T \rightarrow aSb$ sur l'alphabet $\{a, b, c\}$). On ne parle pas d'analyseur LL ou LR. On ne présente pas de théorie générale de l'analyse syntaxique.

Mise en œuvre
On étudie surtout de petits exemples que l'on peut traiter à la main et qui modélisent des situations rencontrées couramment en informatique. On fait le lien avec la définition par induction de certaines structures de données (listes, arbres, formules de logique propositionnelle).

9 Décidabilité et classes de complexité S3-4

On s'intéresse à la question de savoir ce qu'un algorithme peut ou ne peut pas faire, inconditionnellement ou sous condition de ressources en temps. Cette partie permet de justifier la construction, plus haut, d'algorithmes exhaustifs, approchés, probabilistes, etc. On s'appuie sur une compréhension pratique de ce qu'est un algorithme.

Notions	**Commentaires**
Problème de décision. Taille d'une instance. Complexité en ordre de grandeur en fonction de la taille d'une instance. Opération élémentaire. Complexité en temps d'un algorithme. Classe **P**.	Les opérations élémentaires sont les lectures et écritures en mémoire, les opérations arithmétiques, etc. La notion de machine de Turing est hors programme. On s'en tient à une présentation intuitive du modèle de calcul (code exécuté avec une machine à mémoire infinie). On insiste sur le fait que la classe **P** concerne des problèmes de décision.
Réduction polynomiale d'un problème de décision à un autre problème de décision.	On se limite à quelques exemples élémentaires.
Certificat. Classe **NP** comme la classe des problèmes que l'on peut vérifier en temps polynomial. Inclusion $\mathbf{P} \subseteq \mathbf{NP}$.	Les modèles de calcul non-déterministes sont hors programme.
NP-complétude. Théorème de Cook-Levin (admis) : SAT est NP-complet.	On présente des exemples de réduction de problèmes NP-complets à partir de SAT. La connaissance d'un catalogue de problèmes NP-complets n'est pas un objectif du programme.
Transformation d'un problème d'optimisation en un problème de décision à l'aide d'un seuil.	
Notion de machine universelle. Problème de l'arrêt.	

Mise en œuvre
On prend soin de distinguer la notion de complexité d'un algorithme de la notion de classe de complexité d'un problème. Le modèle de calcul est une machine à mémoire infinie qui exécute un programme rédigé en OCaml ou en C. La maîtrise ou la technicité dans des formalismes avancés n'est pas un objectif du programme.

A Langage C

La présente annexe liste limitativement les éléments du langage C (norme C99 ou plus récente) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants à la fin de la première année. Ces éléments s'inscrivent dans la perspective de lire et d'écrire des programmes en C ; aucun concept sous-jacent n'est exigible au titre de la présente annexe.
À l'écrit, on travaille toujours sous l'hypothèse que les entêtes suivants ont tous été inclus : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`. Mais ces fichiers ne font pas en soi l'objet d'une étude et aucune connaissance particulière des fonctionnalités qu'ils apportent n'est exigible.

A.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage C doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

- Typage statique. Types indiqués par le programme lors de la déclaration ou définition.
- Passage par valeur.
- Délimitation des portées par les accolades. Les retours à la ligne et l'indentation ne sont pas signifiants mais sont nécessaires pour la lisibilité du code.
- Déclaration et définition de fonctions, uniquement dans le cas d'un nombre fixé de paramètres.
- Gestion de la mémoire : pile et tas, allocation statique et dynamique, durée de vie des objets.

Définitions et types de base

- Types entiers signés `int8_t`, `int32_t` et `int64_t`, types entiers non signés `uint8_t`, `uint32_t` et `uint64_t`. Lorsque la spécification d'une taille précise pour le type n'apporte rien à l'exercice, on utilise les types signé **`int`** et non signé **`unsigned int`**. Opérations arithmétiques `+`, `-`, `/`, `*`. Opération `%` entre opérandes positifs. Ces opérations sont sujettes à dépassement de capacité. À l'écrit, on élude les difficultés liées à la sémantique des constantes syntaxiques. On ne présente pas les opérateurs d'incrémentation.
- Le type **`char`** sert exclusivement à représenter des caractères codés sur un octet. Notation `'\0'` pour le caractère nul.
- Type **`double`** (on considère qu'il est sur 64 bits). Opérations `+`, `-`, `*`, `/`.
- Type `bool` et les constantes `true` et `false`. Opérateurs `!`, `&&`, `||` (y compris évaluation paresseuse). Les entiers ne doivent pas être utilisés comme booléens, ni l'inverse.
- Opérateurs de comparaison `==`, `!=`, `<`, `>`, `<=`, `>=`.
- Les constantes du programme sont définies par **`const`** *type* *c* **`=`** *v*. On n'utilise pas la directive du préprocesseur **`#define`** à cette fin.

Types structurés

- Tableaux statiques : déclaration par *type* *T*`[`*s*`]` où *s* est une constante littérale entière. Lecture et écriture d'un terme de tableau par son indice *T*`[`*i*`]` ; le langage ne vérifie pas la licéité des accès. Tableaux statiques multidimensionnels.
- Définition d'un type structuré par **`struct`** *nom_s* **`{`***type*$_1$ *champ*$_1$**`;`** … *type*$_n$ *champ*$_n$**`;}`** et ensuite **`typedef struct`** *nom_s* *nom* (la syntaxe doit cependant être rappelée si les étudiants sont amenés à écrire de telles définitions). Lecture et écriture d'un champ d'une valeur de type structure par *v*`.`*champ* ainsi que *v*`->`*champ*. L'organisation en mémoire des structures n'est pas à connaître.
- Chaînes de caractères vues comme des tableaux de caractères avec sentinelle nulle. Fonctions `strlen`, `strcpy`, `strcat`.

Structures de contrôle

- Conditionnelle **`if (`***c***`)`** s_T, **`if (`***c***`)`** s_T **`else`** s_F.
- Boucle **`while (`***c***`)`** *s* ; boucle **`for (`***init***`;`** *fin* **`;`** *incr***`)`** *s*, possibilité de définir une variable dans *init* ; **`break`**.

— Définition et déclaration de fonction, passage des paramètres par valeur, y compris des pointeurs. Cas particuliers : passage de paramètre de type tableau, simulation de valeurs de retour multiples.

Pointeurs et gestion de la mémoire

— Pointeur vers un objet alloué, notation *type*`*` p `= &`v. On considère que les pointeurs sont sur 64 bits.
— Déréférencement d'un pointeur valide, notation `*`p. On ne fait pas d'arithmétique des pointeurs.
— Pointeurs comme moyen de réaliser une structure récursive. Pointeur `NULL`.
— Création d'un objet sur le tas avec `malloc` et `sizeof` (on peut présenter `size_t` pour cet usage mais sa connaissance n'est pas exigible). Libération avec `free`.
— Transtypage de données depuis et vers le type `void*` dans l'optique stricte de l'utilisation de fonctions comme `malloc`.
— En particulier : gestion de tableaux de taille non statiquement connue ; linéarisation de tels tableaux quand ils sont multidimensionnels.

Divers

— Utilisation de `assert` lors d'opérations sur les pointeurs, les tableaux, les chaînes.
— Flux standard.
— Utilisation élémentaire de `printf` et de `scanf`. La syntaxe des chaînes de format n'est pas exigible.
— Notion de fichier d'en-tête. Directive `#include "`*fichier*`.h"`.
— Commentaires `/* ... */` et commentaires ligne `//`

A.2 Éléments techniques devant être reconnus et utilisables après rappel

Les éléments suivants du langage C doivent pouvoir être utilisés par les étudiants pour écrire des programmes dès lors qu'ils ont fait l'objet d'un rappel et que la documentation correspondante est fournie.

Traits généraux et divers

— Utilisation de `#define`, `#ifndef` et `#endif` lors de l'écriture d'un fichier d'en-tête pour rendre son inclusion idempotente.
— Rôle des arguments de la fonction `int main(int argc, char* argv[])` ; utilisation des arguments à partir de la ligne de commande.
— Fonctions de conversion de chaînes de caractères vers un type de base comme `atoi`.
— Définition d'un tableau par un initialisateur `{`t_0`,` t_1`, ...,` t_{N-1}`}`.
— Définition d'une valeur de type structure par un initialisateur `{.`c_1 `=` v_1`, .`c_2 `=` v_2`, ...}`.
— Compilation séparée.

Gestions des ressources de la machine

— Gestion de fichiers : `fopen` (dans les modes `r` ou `w`), `fclose`, `fscanf`, `fprintf` avec rappel de la syntaxe de formatage.
— Fils d'exécution : inclusion de l'entête `pthread.h`, type `pthread_t`, commandes `pthread_create` avec attributs par défaut, `pthread_join` sans récupération des valeurs de retour.
— Mutex : inclusion de l'entête `pthread.h`, type `pthread_mutex_t`, commandes `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`.
— Sémaphore : inclusion de l'entête `semaphore.h`, type `sem_t`, commandes `sem_init`, `sem_destroy`, `sem_wait`, `sem_post`.

B Langage OCaml

La présente annexe liste limitativement les éléments du langage OCaml (version 4 ou supérieure) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

B.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage OCaml doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

— Typage statique, inférence des types par le compilateur. Idée naïve du polymorphisme.
— Passage par valeur.
— Portée lexicale : lorsqu'une définition utilise une variable globale, c'est la valeur de cette variable au moment de la définition qui est prise en compte.
— Curryfication des fonctions. Fonction d'ordre supérieur.
— Gestion automatique de la mémoire.
— Les retours à la ligne et l'indentation ne sont pas signifiants mais sont nécessaires pour la lisibilité du code.

Définitions et types de base

— `let`, `let rec` (pour des fonctions), `let rec` … `and` …, `fun` x y `->` e.
— `let` v `=` e `in` e', `let rec` f x `=` e `in` e'.
— Expression conditionnelle `if` e `then` e_V `else` e_F.
— Types de base : `int` et les opérateurs `+`, `-`, `*`, `/`, l'opérateur `mod` quand toutes les grandeurs sont positives ; exception `Division_by_zero` ; `float` et les opérateurs `+.`, `-.`, `*.`, `/.` ; `bool`, les constantes `true` et `false` et les opérateurs `not`, `&&`, `||` (y compris évaluation paresseuse). Entiers et flottants sont sujets aux dépassements de capacité.
— Comparaisons sur les types de base : `=`, `<>`, `<`, `>`, `<=`, `>=`.
— Types `char` et `string` ; `'`x`'` quand x est un caractère imprimable, `"`x`"` quand x est constituée de caractères imprimables, `String.length`, s`.[`i`]`, opérateur `^`. Existence d'une relation d'ordre total sur `char`. Immuabilité des chaînes.

Types structurés

— n-uplets ; non-nécessité d'un `match` pour récupérer les valeurs d'un n-uplet.
— Listes : type `'a list`, constructeurs `[]` et `::`, notation `[`x`;` y`;` z`]` ; opérateur `@` (y compris sa complexité) ; `List.length`. Motifs de filtrage associés.
— Tableaux : type `'a array`, notations `[|`…`|]`, t`.(`i`)`, t`.(`i`)` `<-` v ; fonctions `length`, `make`, et `copy` (y compris le caractère superficiel de cette copie) du module `Array`.
— Type `'a option`.
— Déclaration de type, y compris polymorphe.
— Types énumérés (ou sommes, ou unions), récursifs ou non ; les constructeurs commencent par une majuscule, contrairement aux identifiants. Motifs de filtrage associés.
— Filtrage : `match` e `with` p_0 `->` v_0 `|` p_1 `->` v_1 … ; les motifs ne doivent pas comporter de variable utilisée antérieurement ni deux fois la même variable ; motifs plus ou moins généraux, notation `_`, importance de l'ordre des motifs quand ils ont des instances communes.

Programmation impérative

— Absence d'instruction ; la programmation impérative est mise en œuvre par des expressions impures ; `unit`, `()`.
— Références : type `'a ref`, notations `ref`, `!`, `:=`. Les références doivent être utilisées à bon escient.
— Séquence `;`. La séquence intervient entre deux expressions.
— Boucle `while` c `do` b `done` ; boucle `for` v `=` d `to` f `do` b `done`.

Divers

- Usage de `begin` … `end`.
- `print_int`, `print_float`, `print_string`, `read_int`, `read_float`, `read_line`.
- Exceptions : levée et filtrage d'exceptions existantes avec `raise`, `try` … `with` … ; dans les cas irrattrapables, on peut utiliser `failwith`.
- Utilisation d'un module : notation $M.f$. Les noms des modules commencent par une majuscule.
- Syntaxe des commentaires, à l'exclusion de la nécessité d'équilibrer les délimiteurs dans un commentaire.

B.2 Éléments techniques devant être reconnus et utilisables après rappel

Les éléments suivants du langage OCaml doivent pouvoir être utilisés par les étudiants pour écrire des programmes dès lors qu'ils ont fait l'objet d'un rappel et que la documentation correspondante est fournie.

Traits divers

- Types de base : opérateur `mod` avec opérandes de signes quelconques, opérateur `**`.
- Types enregistrements mutables ou non, notation {c_0 : t_0; c_1 : t_1; …}, {c_0 : t_0; `mutable` c_1 : t_1; …}; leurs valeurs, notations {c_0 = v_0; c_1 = v_1; …}, $e.c$, $e.c$ <- v.
- Fonctions de conversion entre types de base.
- Listes : fonctions `mem`, `exists`, `for_all`, `filter`, `map`, `iter` du module `List`.
- Tableaux : fonctions `make_matrix`, `init`, `mem`, `exists`, `for_all`, `map` et `iter` du module `Array`.
- Types mutuellement récursifs.
- Filtrage : plusieurs motifs peuvent être rassemblés s'ils comportent exactement les mêmes variables. Notation `function` p_0 -> v_0 | p_1 -> v_1 ….
- Boucle `for` v = f `downto` d `do` b `done`.
- Piles et files mutables : fonctions `create`, `is_empty`, `push` et `pop` des modules `Queue` et `Stack` ainsi que l'exception `Empty`.
- Dictionnaires mutables réalisés par tables de hachage sans liaison multiple ni randomisation par le module `Hashtbl` : fonctions `create`, `add`, `remove`, `mem`, `find` (y compris levée de `Not_found`), `find_opt`, `iter`.
- `Sys.argv`.
- Utilisation de `ocamlc` ou `ocamlopt` pour compiler un fichier dépendant uniquement de la bibliothèque standard.

Gestions des ressources de la machine

- Gestion de fichiers : fonctions `open_in`, `open_out`, `close_in`, `close_out`, `input_line`, `output_string`.
- Fils d'exécution : recours au module `Thread`, fonctions `Thread.create`, `Thread.join`.
- Mutex : recours au module `Mutex`, fonctions `Mutex.create`, `Mutex.lock`, `Mutex.unlock`.