Programme d'informatique
Filière Mathématiques et Physique (MP)
Option Informatique
Première et deuxième années

Table des matières

Introduction au programme

Les objectifs du programme L'enseignement optionnel en informatique est conçu comme un complément à la formation de tronc commun dispensée en classe de MPSI et de MP. Omniprésentes dans les révolutions industrielles et sociales du XXI^{e} siècle, les techniques rattachées à l'informatique connaissent aussi des cycles d'obsolescence rapide et sautent brutalement d'un paradigme à un autre très différent. Voilà pourquoi ce programme privilégie d'opérer par un approfondissement de certains fondamentaux scientifiques de la discipline. Il prépare ainsi de futurs ingénieures et ingénieurs, professeures et professeurs, chercheuses et chercheurs à affronter avec perspective, recul et adaptabilité les défis qu'elles et ils rencontreront dans leur carrière.

Articulation entre le programme de tronc commun et le programme d'option Tantôt le programme d'option complète des chapitres du tronc commun en apportant une profondeur, notamment théorique, sur certains points traités plus sommairement dans le programme initial ; tantôt le programme d'option aborde des thématiques nouvelles et s'attarde à poser les jalons de connaissances et de techniques, plutôt fondamentales, qu'un complément de temps autorise à aborder dans le cadre d'une formation au long cours. La professeure ou le professeur d'informatique de la classe de MPSI, de MP ou de MP* veille donc à la fois à assurer l'intelligibilité de son cours de tronc commun pour les étudiants qui ont choisi de ne pas suivre l'option et à organiser une articulation harmonieuse de son cours d'option avec son cours de tronc commun.

Compétences visées Ce programme amplifie le développement des six grandes compétences identifiées dans le programme de tronc commun :

- **analyser et modéliser** un problème ou une situation, notamment en utilisant les objets conceptuels de l'informatique pertinents (graphe, arbre, automate, 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** un motif textuel, les données d'un problème, ou celles manipulées par un algorithme ou une fonction en utilisant le formalisme approprié (notamment langue française, formule logique, expression régulière) ;
- **mettre en œuvre une solution,** par la traduction d'un algorithme ou d'une structure de données dans le langage de programmation du programme ;
- **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.

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 possibles 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 le langage et la programmation L'enseignement du présent programme repose sur le langage de programmation OCaml dans les perspectives et les limites qui suivent. Après des enseignements centrés sur les langages enseignés dans les classes du secondaire et poursuivis pour partie en tronc commun, ce nouveau langage de nature très différente permet d'approfondir le multilinguisme des étudiants tout en illustrant la diversité des paradigmes de programmation. Le langage OCaml est utilisé en raison de sa capacité à s'ouvrir rapidement à un niveau d'abstraction supérieur et pour sa pertinence dans la manipulation de fonctions ou de structures de données récursives. Les traits impératifs de OCaml sont également présentés : ils permettent en particulier de découvrir des notions centrales, telles que les références, sans excessives difficultés liées au langage employé.

La discipline de programmation mise en place dans le cours de tronc commun reste observée, tout en étant le cas échéant adaptée au cadre de la programmation récursive ou des structure de données immuables. On ne se cantonne pas à écrire des programmes sur papier ; on veille à mettre régulièrement les étudiants en situation de programmer sur machine. Toutefois, la virtuosité dans l'écriture de programmes ou une connaissance exhaustive des bibliothèques de programmation ne sont pas des objectifs de la formation. Au contraire, l'annexe A liste de façon limitative les éléments du langage OCaml qui sont exigibles des étudiants ainsi que ceux auxquels les étudiants sont familiarisés et qui peuvent être attendus à condition que ceux-ci soient accompagnés d'une documentation.

1 Programme de première année

1.1 Programmation récursive

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. Elle permet souvent d'écrire des algorithmes plus élégants et moins laborieux que leurs équivalents en programmation impérative.

Notions	**Commentaires**
Récursivité. Récursivité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples.	On approfondit la présentation purement expérimentale de l'appel récursif d'une fonction à elle-même, vue au premier semestre de tronc commun, en faisant le lien avec le principe de récurrence et les relations d'ordre. On insiste sur le problème de la terminaison et la notion d'ordre bien fondé. Toute théorie générale de la dérécursification est hors programme.
Stratégie diviser pour régner.	On complète, si nécessaire, la présentation de tris élémentaires vue au premier semestre de tronc commun en présentant le tri par partition fusion. On ne se limite pas à des exemples dans lesquels la décomposition et la recomposition des résultats sont évidents.
Analyse de la complexité des algorithmes récursifs.	On étudie sur différents exemples la complexité dans le pire cas. Aucun théorème général de complexité n'est au programme. Sur des exemples, on sensibilise les étudiants à l'existence d'autres analyses de complexité, comme la complexité en moyenne ou la complexité amortie. La notion de classe de complexité d'un problème de décision est hors programme.

1.2 Types récursifs immuables et arbres

Les propriétés sur les structures récursives sont démontrées par des preuves par induction structurelle.

Notions	**Commentaires**
Type récursif de liste.	

Type récursif d'arbre binaire ou d'arbre binaire strict non vide.
 	
Vocabulaire : nœud, nœud interne, racine, feuille, fils, père, hauteur d'un arbre, profondeur d'un nœud, étiquette, sous-arbre.
	`type 'a arbre_binaire = Vide | Noeud of 'a * 'a arbre_binaire * 'a arbre_binaire`
	`type ('a, 'b) abs = Feuille of 'a | Noeud_interne of 'b * ('a, 'b) abs * ('a, 'b) abs`
	La hauteur de l'arbre vide est -1. Relation entre le nombre de nœuds internes et de feuilles d'un arbre binaire strict.

Définition inductive des arbres généraux non vides.	On illustre la notion d'arbre par des exemples, comme : expression arithmétique, arbre préfixe (*trie*), arbre de décision.
Parcours d'arbres. Ordre préfixe, infixe et postfixe.	On peut évoquer le lien avec l'empilement de blocs d'activation lors de l'appel à une fonction récursive.

1.3 Structures de données

Le programme de l'option MP se distingue du tronc commun en ce qu'il appelle la maîtrise du concept de structure de données : il dépasse l'idée que le langage de programmation fournisse une collection appropriée à ses besoins et insiste sur le fait que le développement d'un algorithme aille 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. Bien que la programmation orientée objet ne figure pas dans ce programme, on peut enseigner la notion de structure de données avec cette perspective à l'esprit.

Notions	Commentaires
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.
Distinction entre structure de données mutable et immuable. Référence.	
Tableau, liste, pile, file, dictionnaire.	On complète la présentation purement pratique faite de ces structures dans le cadre du programme de tronc commun en étudiant dans le détail leurs implémentations possibles et en s'attachant à dégager la complexité des opérations associées. On clarifie l'ambiguïté du terme *liste* utilisé en tant que concept en informatique, tableau dynamique dans le langage Python et chaîne de maillons dans le langage OCaml. On présente des problèmes qui peuvent être résolus à l'aide de ces structures. On peut présenter différentes implémentations de la même structure.
Implémentation de la structure immuable de dictionnaire avec un arbre binaire de recherche.	On ne cherche pas à équilibrer les arbres. En lien avec le programme de tronc commun de deuxième année, on peut présenter également une implémentation mutable des dictionnaires avec une table de hachage.
Utilisation d'une structure de données.	Grâce aux bibliothèques, on peut utiliser des structures de données sans avoir à programmer soi-même leur implémentation.

1.4 Syntaxe puis sémantique de la logique propositionnelle

Le but de cette partie est de familiariser progressivement les étudiants avec la différence entre syntaxe et sémantique d'une part et de donner une base de vocabulaire permettant de modéliser une grande variété de situations (par exemple, satisfaction de contraintes, planification, diagnostique, vérification de modèles, etc.).
L'étude des quantificateurs est hors programme.

Notions	Commentaires
Variables propositionnelles, connecteurs logiques, arité. Formules propositionnelles, définition inductive, 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.
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. Problème SAT.	Lien entre forme normale disjonctive complète et table de vérité.

2 Programme de deuxième année

2.1 Applications des arbres

On aborde les arbres comme support de pensée qui permet de donner du sens et de raisonner sur le flot de contrôle qui s'observe dans la mise en œuvre de stratégies algorithmiques ou de structures de données élaborées.

Notions	Commentaires
Retour sur trace (*backtracking*).	On présente la notion à travers quelques exemples sans théorie générale (par exemple, problème des huit reines ou résolution d'un sudoku). La notion de retour sur trace est réinvestie par l'étudiant à l'occasion de l'étude de l'algorithme min-max présenté en tronc commun.
File de priorité.	Implémentation de la structure mutable de file de priorité bornée à l'aide d'un tas stocké dans un tableau. On illustre l'intérêt d'une file de priorité pour améliorer l'implémentation sommaire de l'algorithme de Dijkstra vue dans le cadre du programme de tronc commun. On présente le tri par tas d'un tableau. Pour l'algorithme de Kruskal, l'utilisation d'une file de priorité est une alternative intéressante au tri des arêtes lorsqu'elles sont stockées dans un tableau.

2.2 Graphes avancés

Le vocabulaire et les définitions relatives aux graphes, orientés ou non, tels que sommets, arcs, arêtes, degrés, ont été présentés dans le cadre du programme d'informatique de tronc commun.

Notions	Commentaires
Notion de parcours (sans contrainte). Parcours en largeur, en profondeur. Notion d'arborescence d'un parcours.	On approfondit la notion de parcours de graphe dont l'étude a été entamée dans le cadre du programme de tronc commun et on en détaille quelques applications : par exemple, recherche et construction d'un cycle, calcul des composantes connexes, bicolorabilité, états accessibles d'un automate. On étudie l'intérêt d'un type de parcours dans le cadre d'applications. Par exemple : plus courts chemins dans un graphe à distance unitaire ou tri topologique dans un graphe acyclique orienté. On implémente ces parcours à l'aide d'une représentation du graphe en listes d'adjacence et on discute leur complexité.
Arbre couvrant dans un graphe pondéré.	Algorithme de Kruskal. Les détails d'implémentation sont laissés à l'appréciation du professeur. On fait le lien avec la notion d'algorithme glouton étudiée dans le programme de tronc commun.
Graphe biparti. Couplage.	Recherche d'un couplage de cardinal maximum dans un graphe biparti par des chemins augmentants. On se limite à une approche élémentaire. L'algorithme de Hopcroft-Karp n'est pas au programme.

2.3 Déduction naturelle pour la logique propositionnelle

Il s'agit de présenter les preuves comme permettant de pallier deux problèmes de la présentation du calcul propositionnel faite en première année : 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.
On s'abstient d'implémenter ces règles. L'ambition est d'être capable d'écrire de petites preuves dans ce système.

Notions	Commentaires
Déduction naturelle. 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.).

2.4 Langages et automates

On introduit les expressions régulières comme formalisme dénotationnel pour spécifier un motif dans le cadre d'une recherche textuelle et les automates comme 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
Alphabet, mot, préfixe, suffixe, facteur, sous-mot. Langage comme ensemble de mots sur un alphabet.	Le mot vide est noté ε.
Opérations régulières sur les langages (union, concaténation, étoile de Kleene). Définition inductive des langages réguliers. Expression régulière. Dénotation d'une expression régulière.	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.
Automate fini déterministe. État accessible, co-accessible. Automate émondé; automate complet. Langage reconnu par un automate.	
Automate fini non déterministe. Transition spontanée (ou ε-transition). Déterminisation d'un automate non déterministe.	On aborde l'élimination des ε-transitions 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 $xy^*z \subseteq L$.

A 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.

A.1 Traits et éléments techniques à connaitre

Les éléments et notations suivants du langage OCaml doivent pouvoir être compris et utilisés par les étudiants dès la fin de la première année 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 ; `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 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.
— 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 sont exhaustifs, ils 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` (on rappelle, quand cela est utile au problème étudié, que les deux bornes sont atteintes).

Divers

— Usage de `begin` … `end`.

— Exceptions : `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.

A.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.

Définition et types de base

— Types de base : opérateur `mod` avec opérandes de signes quelconques, opérateur `**`.
— 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 chaines.
— Fonctions de conversion entre types de base.
— `print_int`, `print_string`, `print_float`.

Types structurés et structures de données

— Listes : les fonctions `mem`, `exists`, `for_all`, `filter`, `map` et `iter` du module `List`.
— Tableaux : type `'a array`, notations `[|...|]`, t`.(`i`)`, t`.(`i`) <-` v ; les fonctions suivantes du module `Array` : `length`, `make`, `make_matrix`, `init`, `copy`, `mem`, `exists`, `for_all`, `map` et `iter`.
— Types enregistrements immuables et mutables, notations associées.
— Types mutuellement récursifs.
— Piles et files mutables : fonctions `create`, `is_empty`, `push` et `pop` des modules `Queue` et `Stack`.
— Dictionnaires mutables réalisés par tables de hachage sans liaison multiple ni randomisation par le module `Hashtbl` : fonctions `create` (on élude toute considération sur la taille initiale), `add`, `remove`, `mem`, `find`, `find_opt` et `iter`.