

Algorithmique-Programmation

IAP1

Catherine Dubois

Bloc Algorithmique-Programmation (IAP)

- ▶ Fondements de la programmation (IAP1)
- ▶ Programmation impérative et structures de données simples (IAP2)
- ▶ Structures de données avancées, Modularité (IAP3)

suivi du **projet informatique (IPI)** (3 mois de début mars à fin mai)

- **Fondements de la programmation (IAP1)**

Concepts fondamentaux de la programmation (par exemple, types, récursion)

Initiation au style fonctionnel

Initiation au développement de logiciels fiables (spécification, preuve)

Cours, TD (4 groupes), TP (8 demi-groupes + 1), présence obligatoire et contrôlée

Langage utilisé : OCaml (seule la partie purement fonctionnelle)

Evaluation : projet individuel (dossier + soutenance) (3) + contrôle continu (2) + contrôle écrit (3)

Responsable : C. Dubois, dubois@ensiie.fr, bureau 209

- **Programmation impérative et structures de données simples (IAP2)**

Apprendre un langage de programmation impératif

Manipulation de pointeurs

Utilisation d'un outil de débogage

Structures de données simples : tableaux, listes, piles, files

Cours, TD (4 groupes), TP (8 demi-groupes)

Langage utilisé : C

Evaluation : projet individuel (dossier + soutenance) + contrôle écrit + contrôle continu

Responsable : R. Rioboo, rioboo@ensiie.fr

- **Structures de données avancées, Modularité (IAP3)**

Structures de données complexes : graphes, arbres, hachage ...

Savoir évaluer la complexité d'un algorithme et choisir la structure de données adaptée

Conception et développement modulaire

Cours, TD + TP (4 groupes)

Langages utilisés : C, OCaml

Evaluation : contrôle continu + contrôle sur machine

Responsable : J. Forest, forest@ensiie.fr

- **Projet informatique (IPI)**

Mettre en pratique les notions vues lors de IAP, en particulier modularité, structures de données.

Programmes de plus grande taille, travail en équipe

Techniques du Génie Logiciel, présentation et utilisation d'outils orientés GL (CVS gestion de versions)

Langage utilisé : C

Organisation : par équipe de 6 à 8 étudiants.

Projet encadré par un enseignant.

Evaluation : rapports et travaux à rendre selon planning imposé + soutenances

Quelques infos concernant l'utilisation des ordinateurs

- Accès aux machines : en libre accès quand les salles ne sont pas occupées par des TPs
- login, email, accès Web : **Initiation au mel et réseau**
- Accès depuis les résidences : voir les élèves, l'équipe systèmes.
- Prise en main des machines : **TPM IAP**
⇒ *Initiation à Unix et à l'éditeur de textes Emacs*

Introduction à la programmation
ENSIIE 1A
Septembre 2009
C. Dubois

- Qu'est-ce qu'un programme ?

programme = codage d'un algorithme, solution d'un problème, dans un langage de programmation

- La programmation consiste à *modéliser* un problème du monde réel sous une forme symbolique (ou numérique) en vue de le faire résoudre par un ordinateur

problèmes décidables / problèmes indécidables

- On utilise un langage de programmation pour décrire la solution du programme (comment la calculer).

Un langage de programmation a une syntaxe et une sémantique bien définies

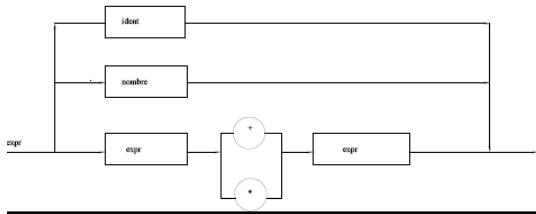
Syntaxe : ensemble de règles lexicales et de règles de grammaire qui précisent comment écrire les programmes

description par exemple à l'aide de diagrammes ou d'une grammaire BNF et/ou d'expressions régulières

`nombre = 0 | [-][1..9][0..9]*`

`expr ::= nombre | expr + expr | expr * expr | idf`

Dans IAP, on se contentera le plus souvent de descriptions informelles



Erreur de syntaxe : une de ces règles n'est pas respectée

exemples : -0 n'est pas un nombre bien écrit,

+ 10 2 n'est pas une expression autorisée

le chyen n'est pas un mot correct du français,

le chien dans la niche est n'est pas correcte syntaxiquement

Outil de vérification associé : analyseur lexical, analyseur syntaxique
(génération possible à partir de la grammaire : Lex, Yacc)

Sémantique : décrit le sens de chacune des constructions du langage

En particulier elle décrit le sens des opérateurs

Par exemple $+$ est interprété comme l'addition de 2 entiers

\wedge est interprété comme la concaténation de deux chaînes de caractères

... \Rightarrow **la sémantique décrit comment évaluer une expression**

Description en langage naturel ou mieux description formelle (dans un langage à base mathématique)

Ici pour l'instant en langage naturel le moins ambigu possible

Comment passe-t-on de la description d'un problème du monde réel au programme pour le résoudre ?

Comment passer de l'énoncé du problème à un programme de bonne qualité ?

On entend par *bonne qualité* :

- ▶ Lisible
- ▶ Correct c'est-à-dire qui résoud effectivement le problème posé
- ▶ Robuste
- ▶ Facile à réutiliser ou à modifier. En effet, un logiciel, souvent coûteux à produire, est en général utilisé pendant longtemps et on est souvent amené à le modifier, à l'adapter à de nouveaux besoins : on parle alors de *maintenance* de logiciel. Ainsi un programme bien documenté, lisible, bien organisé sera plus facile à modifier par la suite.

Quelles sont les étapes du développement d'un programme ?

On part d'un problème qui se pose dans le monde réel.

Ce problème, exposé dans une langue naturelle, est généralement flou, imprécis, mal posé.

● **Première étape : la spécification.** description complète et rigoureuse du problème à résoudre

QUOI ?

Le langage de spécification doit être le plus proche possible des mathématiques, pour :

- ▶ éviter toute ambiguïté,
- ▶ servir de support à des démonstrations

L'étape de spécification permet de découvrir puis résoudre les points susceptibles de poser problème, au plus tôt, c'est-à-dire avant la réalisation même du logiciel.

Il existe différentes méthodes de spécification (cours 2A)

Ici des énoncés informels de problèmes les plus clairs et les moins ambigus possible

- **Deuxième étape : la recherche d'un algorithme pour résoudre le problème**

On cherche ici un algorithme qui répond au problème

COMMENT ?

L'algorithme est généralement décrit dans un langage assez mathématique où l'on supposera que l'on dispose de fonctions élémentaires élaborées.

Ici langage pour les algorithmes : inspiré de notre langage de programmation

•Troisième étape : la transcription de l'algorithme dans un langage de programmation

Codage de l'algorithme dans un langage de programmation utilisable sur la machine

Proximité des langages de programmation avec les 2 langages précédents facile le passage de la spécification au programme.

Bien sûr, l'étape de programmation ne peut ignorer complètement les caractéristiques de la machine (entrées-sorties)

Le programme est ensuite compilé c'est-à-dire traduit en langage machine

•**Quatrième étape : la validation du programme**

s'assurer plus ou moins fortement que le programme développé produit les résultats attendus.

Les étapes précédentes, si elles ont été faites avec rigueur et raisonnement, ont dues déjà éliminer un certain nombre d'erreurs.

Mais on ne peut affirmer que le programme est correct (par rapport à sa spécification)

Attention, la correction d'un programme est toujours relative à une spécification donnée

Deux techniques différentes mais complémentaires : le test et la preuve.

Le test :

essayer le programme sur divers exemples et comparer les résultats calculés par le programme avec ceux attendus (oracle)

Les exemples envisagés, appelés cas de test, correspondent par exemple à :

- ▶ des données nominales (considérées comme normales),
- ▶ exceptionnelles, aux limites
- ▶ ou encore erronées.

Paradoxalement, c'est lorsqu'un des cas de test révélera une erreur que la méthode sera considérée comme positive.

Avec le test seul, on ne peut pas dire :

mon programme est bon

mais on peut dire

le programme n'est pas en erreur sur les cas de test envisagés

Plus le nombre de cas de test est élevé, et plus ceux-ci sont pertinents, plus on se rapproche de l'affirmation *mon programme est bon* .

la preuve :

démontrer mathématiquement que le programme satisfait sa spécification.

spécification formelle du programme (description mathématique, logique)

+ un modèle mathématique de ce que fait le programme

des outils d'aide à la preuve existent pour assister et automatiser tout ou une partie de travail

(une propriété est-elle vraie ? **Problème indécidable**)

●**Exemple:** calcul du nombre de digits de la représentation d'un nombre en base 2

Description informelle : soit x un entier naturel, combien faut-il de digits pour représenter x en base 2 ?

Spécification : plus formellement, écrire un programme qui calcule les valeurs de la fonction lg définie de \mathbb{N} dans \mathbb{N} telle que $lg(0) = 1$ et $\forall x \in \mathbb{N}^*, 2^{lg(x)-1} \leq x < 2^{lg(x)}$.

Solutions algorithmiques : il y en a plusieurs !

1. une première solution itérative : soit x un entier donné, calculer les puissances successives de 2 ($2^1, 2^2, \dots, 2^i, \dots$), i.e. les termes de la suite $(2^n)_{n \in \mathbb{N}^*}$
2. une deuxième solution : si on sait calculer $lg(x)$, on sait calculer $lg(2x)$ et $lg(2x + 1)$ qui valent $lg(x) + 1$.
On déduit donc que $\forall x > 1, lg(x) = lg(x \div 2) + 1$

Solution récursive

```
let rec lg x = if x <= 1 then 1
               else lg (x/2) + 1;;
```

Validation :

Le programme satisfait-il la spécification ?

A-t-on $lg(0) = 1$ et $\forall x \in \mathbb{N}^*, 2^{lg(x)-1} \leq x < 2^{lg(x)}$???????

1. par la preuve :

⇒ principe de récurrence généralisée sur \mathbb{N}^*

2. par le test :

On cherche des données de test pour *passer par tous les cas de la spécification* : tests nominaux

⇒ DT_1 : 0 (on attend le résultat 1)

⇒ DT_2 : un entier non nul puissance de 2, 16 par exemple (on attend 5)

⇒ DT_3 : un entier non nul non puissance de 2, 231 par exemple (on attend 8)

Contenu du cours

Pour faire apprentissage de la programmation raisonnée : Ocaml

<http://caml.inria.fr/>

grande puissance d'expression, sûreté du langage, pédagogie et portabilité,
transposition aisée à d'autres langages

Contenu du cours : les points principaux

- ▶ présentation rapide du langage OCaml, prise en main immédiate (découverte ou révision) : TPs
- ▶ preuve de programmes
- ▶ types sommes, structures inductives
- ▶ sémantique, typage

La page du cours : www.ensiie.fr/~dubois/pageIAP1.html

accessible depuis l'intranet cours en ligne/Informatique 1A/IAP1.

(comptes génériques : login : eleve mot de passe : !ensiie!)

Spécification et Conception

La spécification se matérialise en une interface

Ecrire **AVANT** toute chose

Indépendante du corps de l'algorithme (du COMMENT)

Une interface est à fournir pour chaque fonction du programme
(obligatoirement en commentaire dans les programmes OCaml)

De plus, on pensera aux tests dès la spécification : les données de tests figurent dans l'interface

```
(* Interface nom de la fonction
type : son type
arguments arg1 : un texte
      ...
      argn : un texte
pré : precondition
post : postcondition
raises : nom des exceptions pouvant être déclenchées
test : tests nominaux (un test pour chaque cas de la spécif
      et quelques tests "limites"
*)
```

La précondition exprime la condition sous laquelle on obtiendra un résultat qui satisfait la précondition.

Le champ `raises` indique ce qui se passe quand les arguments ne satisfont pas la précondition.

Exemple1 On veut concevoir une fonction qui calcule la racine carrée positive d'un nombre réel.

```
(* Interface
racine  : float -> float
arguments x : flottant dont on veut calculer la racine
pré : x >= 0
post : resultat * resultat = x
      ou encore (racine x) * (racine x) = x
test : racine 25.0 (*5*), racine 51.34 (* *) ,
      racine (-25. (*erreur*))
*)
```

Exemple2

```
(* Interface
augmenter  : int -> int
arguments x : entier naturel
contexte : square : int -> int calcule le carré d'un n
pré : x >=0
post : resultat > square x
      ou encore (augmenter x) > (square x)
test : augmenter 10 (*plus gd que 10*), augmenter 0,
      augmenter (-4) (* erreur*)
*)
```

Exemple3

```
(* Interface
lendemain : date -> date
arguments d : date dont on veut calculer le lendemain
pré : date valide
post : lendemain d est la date qui désigne le
      jour suivant la date d
      (selon le calendrier grégorien)
test : etc.
*)
```

date est ici un type défini auparavant.

Algorithmes et programmes : conception descendante

Quel langage utiliser pour décrire l'algorithme ?

⇒ **Notre réponse : OCaml ou pseudo-OCaml**

succession d'algorithmes de plus en plus détaillés qui aboutiront au programme final

dans les premiers algorithmes : *trous* (phrases ou des morceaux de phrases en langage naturel)

Exemple : calculer la date du lendemain

Premier niveau d'algorithme : pas un programme exécutable

```
let lendemain jour =  
if jour est le dernier jour du mois  
then passer au 1er du mois suivant  
else passer au jour suivant
```

Deuxième niveau d'algorithme : toujours pas exécutable

```
let lendemain jour =  
if jour est le dernier jour du mois  
then  if mois = decembre  
       then passer au 1er janvier de l'annee suivante  
       else passer au 1er du mois suivant, même année  
else  passer au jour suivant
```

etc ... vers le programme final (en TP)