Logique

1ère année

Haute École de Bruxelles — École Supérieure d'Informatique

Année académique 2013 / 2014

Exemple

Rédiger une marche à suivre détaillée qui explique comment additionner deux fractions :

1. Rechercher le dénominateur commun des deux fractions
2. Mettre chaque fraction au même dénominateur
3. Additionner les numérateurs des deux fractions, ce qui donne le numérateur de la somme
4. Simplifier la fraction obtenue

Encore très imprécis.
Un algorithme proche d'un langage de programmation ne devrait mentionner que les opérations élémentaires de calcul telles que $+$,

Exemple plus proche d'un programme écrit dans un langage compréhensible par l'ordinateur

1. Prendre connaissance du premier numérateur et du premier dénominateur ;
2. Prendre connaissance du second numérateur et du second dénominateur ;
3. Multiplier les deux dénominateurs pour obtenir le dénominateur commun ;
4. Multiplier le premier numérateur par le second dénominateur et le second numérateur par le premier dénominateur ;
5. Additionner ces deux produits pour obtenir le numérateur du résultat ;

Langage

Le français n'est pas adapté à la description de problèmes au contenu mathématique ou scientifique.
Ceci dit, le français peut toujours être utilisé dans la formalisation d'une première approche de la résolution d'un problème devant être traduit ensuite en algorithme puis en programme.

Le pseudo-code

français : trop grande richesse de ce langage en comparaison à la pauvreté (mais aussi la rigueur) du langage compris par la machine.

Le **pseudo-code** ou *Langage de Description des Algorithmes* (LDA en abrégé) est un langage formel et symbolique utilisant :

- des **noms symboliques** destinés à représenter les objets sur lesquels s'effectuent des actions ;
- des **opérateurs symboliques** ou des mots-clés traduisant les opérations primitives exécutables par un exécutant donné ;
- des **structures de contrôle** types.

Le pseudo-code

Un algorithme *idéal*, appelé **algorithme général**, exprimé en pseudo-code, devrait se situer à mi-chemin entre la démarche globale exprimée dans un langage naturel (langue française) ou structuré (ordinogramme) et l'algorithme ultime, c'est-à-dire le **programme**, exprimé en langage de programmation.

À partir du moment où des conventions sont prises dans un contexte bien déterminé, il convient que **tous** respectent ces conventions.

Variables et type

- les opérations que l'ordinateur devra exécuter portent sur des éléments qui sont les **données** du problème
- lorsqu'on attribue un **nom** et un **type** à ces données, on parle alors de **variables**
- dans un algorithme, une variable conserve toujours son nom et son type, mais peut changer de **valeur**
- le **nom** d'une variable permet de la caractériser et de la reconnaitre
- le **type** d'une variable décrit la nature de son contenu

Types autorisés

Dans un premier temps, les seuls **types** utilisés sont les suivants :

entier	pour les nombres entiers
réel	pour les nombres réels
caractère	pour les différentes lettres et caractères (par exemple ceux qui apparaissent sur un clavier : 'a', '1', '#', etc. . .)
chaine	pour les variables contenant plusieurs caractères ou aucun (la chaine vide) (par exemple : "Bonjour", "Bonjour le monde", "a", "", etc.)
booléen	les variables de ce type ne peuvent valoir que **vrai** ou **faux**

Exercices

Quel(s) type(s) de données utiliseriez-vous pour représenter

- une date du calendrier ?
- un moment dans la journée ?
- le prix d'un produit en grande surface ?
- votre nom ?
- vos initiales ?
- votre adresse ?

Déclaration de variables

La déclaration d'une variable est l'instruction qui définit son nom et son type.

```
num1, num2 : entiers
```

L'ensemble des instructions de la forme

```
variable1, variable2,... : type
```

forme la partie d'un algorithme nommée **déclaration des variables**.

La déclaration des informations apparaitra toujours en début d'algorithme, ou dans un bloc annexe appelé **dictionnaire des variables** ou encore **dictionnaire des données**.

Déclaration de variables

Par exemple, pour l'algorithme des fractions, la déclaration des informations pourrait être la suivante :

```
num1, den1, num2, den2, numRes, denRes : entiers
```

avec la signification suivante :

1. *num1 (num2)* : le numérateur de la première (seconde) fraction ;
2. *den1 (den2)* : le dénominateur de la première (seconde) fraction ;
3. *numRes (denRes)* : le numérateur (dénominateur) du résultat.

Déclaration de variables

Attention, lors de la déclaration d'une variable, celle-ci n'a pas de valeur ! Nous verrons plus loin que c'est l'instruction d'**affectation** qui va servir à donner un contenu aux variables déclarées. En logique, nous n'envisageons pas d'*affectation par défaut* consistant à donner une valeur initiale de façon automatique aux variables déclarées (par exemple 0 pour les variables numériques, comme c'est le cas dans certains langages informatiques).

Comment nommer correctement une variable ?

Un nom

- suffisamment court tout en restant explicite
- qui ne prête pas à confusion
- qui tient compte des limitations imposées par les langages de programmation

Comment nommer correctement une variable ?

Voici quelques règles et limitations traditionnelles dans les langages de programmation :

- Un nom de variable est généralement une suite de caractères alphanumériques d'un seul tenant (pas de caractères blancs) et ne commençant jamais par un chiffre. Ainsi *x1* est correct mais non *1x*.

Comment nommer correctement une variable ?

- Pour donner un nom composé à une variable, on peut utiliser le « tiret bas » ou *underscore* (par ex. premier_numérateur) mais on déconseille d'utiliser le signe « – » qui est plutôt réservé à la soustraction. Ainsi, dans la plupart des langages, *premier-numérateur* serait interprété comme la soustraction des variables *premier* et *numérateur*. (Signalons que le tiret « – » est autorisé en Cobol, où il récupère son rôle arithmétique s'il est précédé et suivi d'au moins un blanc).

Comment nommer correctement une variable ?

- Une alternative à l'utilisation du tiret bas pour l'écriture de noms de variables composés est la notation « chameau » (*camelCase* en anglais), qui consiste à mettre une majuscule au début des mots (généralement à partir du deuxième), par exemple *premierNombre* ou *dateNaissance*.
- Les indices et exposants sont proscrits (par ex. x_1, z_6 ou m^2)

Comment nommer correctement une variable ?

- Les mots clés du langage sont interdits (par exemple **for**, **if**, **while** pour Java et Cobol) et on déconseille d'utiliser les mots-clés du pseudo-code (tels que **lire**, **afficher**, **pour**. . .)
- Certains langages n'autorisent pas les caractères accentués (tels que *à*, *ç*, *ê*, *ø*, etc.) ou les lettres des alphabets non latins (tel que Δ) mais d'autres oui ; certains font la distinction entre les minuscules et majuscules, d'autres non. En logique, nous admettons dans noms de variables les caractères accentués du français, par ex. : durée, intérêts, etc.

Exercices

Déclarer le(s) variable(s) permettant de représenter

- la date d'anniversaire de quelqu'un.
- l'heure de début, l'heure de fin et l'objet d 'un rendez-vous.

Opérateurs et expressions

Les opérateurs agissent sur les variables et les constantes pour former des **expressions**.
Une expression est donc une combinaison **cohérente** de variables, de constantes et d'opérateurs, éventuellement accompagnés de parenthèses.

Opérateurs arithmétiques élémentaires

Ce sont les opérateurs binaires bien connus :

$+$	addition
-	soustraction
*	multiplication
$/$	division réelle
DIV	division entière
MOD	reste de la division entière

Opérateurs arithmétiques élémentaires

Ils agissent sur des variables ou expressions à valeurs entières ou réelles.
Plusieurs opérateurs peuvent être utilisés pour former des expressions plus ou moins complexes, en tenant compte des règles de calcul habituelles, notamment la priorité de la multiplication et de la division sur l'addition et la soustraction.
Il est aussi permis d'utiliser des parenthèses.

Fonctions mathématiques complexes

L'élévation à la puissance sera notée ** ou ^ .

Pour la racine carrée d'une variable x nous écrirons $\sqrt{x}$. Attention, pour ce dernier, de veiller à ne l'utiliser qu'avec un radicant positif !

Si nécessaire, on se permettra d'utiliser les autres fonctions mathématiques sous leur forme la plus courante dans la plupart des langages de programmation (exemples : $sin(x)$, $tan(x)$, $log(x)$, $exp(x)$, ...)

Fonctions mathématiques complexes

Exemple : $(-b + \sqrt{(b**2 - 4 * a * c)})/(2 * a)$

Mais on peut aussi accepter la notation mathématique usuelle $\frac{-b+\sqrt{b^2-4*a*c}}{2*a}$

Pourquoi ne pas avoir écrit « *4ac* » et « *2a* » ?

Opérateurs de comparaison

Ces opérateurs agissent généralement sur des variables numériques ou des chaines et donnent un résultat booléen.

$=$	égal
$<>$ ou $\neq$	différent de
$<$	(strictement) plus petit que
$>$	(strictement) plus grand que
$\leq$	plus petit ou égal
$\geq$	plus grand ou égal

Pour les chaines, c'est l'ordre alphabétique qui détermine le résultat (par exemple *"milou"* $<$ *"tintin"* est **vrai** de même que *"assembleur"* $\leq$ *"java"*)

Opérateurs logiques

Ils agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type.

NON	négation
ET	conjonction logique
OU	disjonction logique

Opérateurs logiques

Pour rappel, *cond1 ET cond2* n'est vrai que lorsque les deux conditions sont vraies.

cond1 OU cond2 est toujours vrai, sauf quand les deux conditions sont fausses.

Veillez à mettre des parenthèses dans le cas de combinaisons de ET et de OU : *(cond1 ET cond2) OU cond3* étant différent de *cond1 ET (cond2 OU cond3).*
En cas d'oubli de parenthèses, il faudra se rappeler que *ET* est prioritaire sur le *OU*.

Opérateurs logiques

Pour rappel aussi, pour un booléen *ok* :
ok = *faux* est équivalent à *NON ok*,
ok = *vrai* est équivalent à *ok* et
NON NON ok est équivalent à *ok*.
Dans les trois cas, nous préconiserons la seconde écriture.

Évaluation complète et court-circuitée

On définit deux modes d'évaluation des opérateurs *ET* et *OU* :

- l'évaluation *complète*
- l'évaluation *court-circuitée*

Dans le cadre de ce cours, nous opterons pour la deuxième interprétation.

Exemple : considérons l'expression $n \neq 0$ *ET* $m/n > 10$.

Manipuler les chaines

Pour les chaines, nous allons introduire quelques notations qui vont nous permettre de les manipuler plus facilement.

- *long(maChaine)* donne la taille (le nombre de caractères) de la chaine *maChaine*.
- *car(maChaine,pos)* donne le caractère en position *pos* (à partir de 1) dans la chaine *maChaine*.
- *concat(maChaine1,maChaine2)* concatène les chaines *maChaine1* et *maChaine2*. (ex : *concat("Bon","jour")* donne *"Bonjour"*)

Manipuler les caractères

Introduisons également quelques notations pour les caractères.

- *chaine(car)* transforme le caractère *car* en une chaine de taille 1.
- *estLettre(car)* est vrai si le caractère *car* est une lettre (idem pour *estChiffre*, *estMajuscule*, *estMinuscule*).
- *majuscule(car)* donne la majuscule de la lettre *car* (idem pour *minuscule*).
- *position(car)* donne la position de *car* dans l'alphabet (ex : *numLettre('E')* donne 5, idem pour *numLettre('e')*).
- *lettre(num)* l'inverse de la précédente (ex : *lettre(4)* donne le caractère 'D').

Affectation externe

L'**affectation externe** est la primitive qui permet de recevoir de l'utilisateur, au moment où l'algorithme se déroule, une ou plusieurs valeur(s) et de les affecter à des variables en mémoire.

```
lire liste_de_variables_à_lire
```

Affectation interne

On parle d'affectation interne lorsque la valeur d'une variable est « calculée » par l'exécutant de l'algorithme lui-même à partir de données qu'il connait déjà :

```
nomVariable ← expression
```

Par exemple :

```
somme ← nombre1 + nombre2
denRes ← den1 * den2
cpt ← cpt + 1
delta ← b**2 – 4*a*c
test ← a < b                          // pour une variable logique
maChaine ← "Bon"
uneChaine ← concat(maChaine, "jour")
```

Affectation

Il est de règle que le résultat de l'expression à droite du signe d'affectation ($\leftarrow$) soit de même type que la variable à sa gauche. On tolère certaines exceptions :

- *varEntière* $\leftarrow$ *varRéelle* : dans ce cas le contenu de la variable sera la valeur **tronquée** de l'expression réelle. Par exemple si « *nb* » est une variable de type entier, son contenu après l'instruction « *nb* $\leftarrow 15/4$ » sera 3
- *varRéelle* $\leftarrow$ *varEntière* : ici, il n'y a pas de perte de valeur.
- *varChaine* $\leftarrow$ *varCaractère* :
  équivalent à *varChaine* $\leftarrow$ *chaine(varCaractère)*.
  Le contraire n'est évidemment pas accepté.

Affectation

- Seules les variables déclarées peuvent être affectées, que ce soit par l'affectation externe ou interne !
- Nous ne mélangerons pas la déclaration d'une variable et son affectation interne dans une même ligne de code, donc pas d'instructions hybrides du genre x $\leftarrow$ *2 : entier* ou encore x : entier(0).
- Pour l'affectation interne, toutes les variables apparaissant dans l'*expression* doivent avoir été affectées préalablement. Le contraire provoquerait un arrêt de l'algorithme.

Communication des résultats

L'instruction de communication des résultats consiste à donner à l'extérieur (donc à l'utilisateur) la valeur d'un résultat calculé au cours de l'exécution de l'algorithme. Nous écrirons :

afficher *expression* ou *liste de variables séparées par des virgules*

qui signifie que la valeur d'une expression (ou celles des différentes variables mentionnées) sera fournie à l'utilisateur (par exemple par un affichage à l'écran ou par impression sur listing via l'imprimante, etc. . .).

Comme pour l'affectation interne, on ne peut *afficher* que des expressions dont les variables qui la composent ont été affectées préalablement.

Structure générale d'un algorithme

La traduction d'un algorithme en pseudo-code constituera le contenu d'un *module*. Un module contient donc la solution algorithmique d'un problème donné (ou d'une de ses parties). Sa structure générale sera la suivante :

module *nomDuModule*()
    déclaration des variables et constantes utilisées dans le module
    lecture des données
    instructions utilisant les données lues
    communication des résultats
fin module

Comme pour les variables, le nomDuModule devra être approprié au contenu.

Fraction(suite)

récrivons l'algorithme d'addition de fractions décrit en début de chapitre :

```
module additionnerFractions()
    num1, den1, num2, den2, numRes, denRes : entiers
    lire num1, den1, num2, den2
    denRes ← den1 * den2
    numRes ← num1*den2 + num2*den1
    afficher numRes, "/", denRes
fin module
```

Remarque : rappelons que la fraction affichée n'est sans doute pas simplifiée. Nous n'avons pas encore tous les atouts suffisants pour réaliser cela à ce niveau. Patience !

Commenter un algorithme

On n'insistera jamais assez sur la nécessité de **documenter** un algorithme en y insérant des **commentaires** judicieux, clairs et suffisants ! Un commentaire est un texte placé dans l'algorithme et destiné à faciliter au maximum la compréhension d'un algorithme par le lecteur (parfois une autre personne, mais aussi souvent l'auteur qui se perd dans son propre texte lorsqu'il s'y replonge après une interruption). Ces commentaires (introduits par « // ») seront bien entendu ignorés par l'exécutant de l'algorithme.

Commenter fraction

```
// Lit les contenus de 2 fractions et affiche leur somme
module additionnerFractions()
    num1, den1, num2, den2, numRes, denRes : entiers
    lire num1, den1, num2, den2
    denRes ← den1 * den2                    // calcul du dénominateur
    numRes ← num1*den2 + num2*den1                    // calcul du
numérateur
    // la fraction n'est sans doute pas simplifiée
    afficher numRes, "/", denRes
fin module
```

Commentaires

Noter qu'un excès de commentaires peut être aussi nuisible qu'un trop-peu pour la compréhension d'un algorithme. Par exemple, un choix judicieux de noms de variables peut s'avérer bien plus efficace que des commentaires superflus. Ainsi, l'instruction

```
nouveauCapital ← ancienCapital * (1 + taux / 100)
```

dépourvue de commentaires est bien préférable aux lignes suivantes :

```
c1 ← c0 * (1 + t / 100)
// calcul du nouveau capital
// c1 est le nouveau capital, c0 est l'ancien capital, t est le taux
```

Nous prendrons l'habitude de commenter chaque module en précisant ce qu'il fait

Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés. La liste des constantes utilisées dans un algorithme apparaitra dans la section déclaration des variables sous la forme suivante :

constante PI = 3,14
constante ESI = "École Supérieure d'Informatique"

Il est inutile de spécifier leur type, celui-ci étant défini implicitement par la valeur de la constante.

Énumération

Parfois, une variable ne peut prendre qu'un ensemble fixe et fini de valeurs. Par exemple une variable représentant une saison ne peut prendre que quatre valeurs (HIVER, PRINTEMPS, ÉTÉ, AUTOMNE). On va l'indiquer grâce à l'énumération qui introduit un **nouveau type** de donnée.

> **énumération** Saison { HIVER, PRINTEMPS, ÉTÉ, AUTOMNE }

Il y a deux avantages à cela : une indication claire des possibilités de la variable lors de la déclaration et une lisibilité du code grâce à l'utilisation des valeurs explicites.

Énumération : exemple

Par exemple,

```
// Lit une saison et affiche sa particularité
module particularitéSaisonnière()
    uneSaison : Saison
    lire uneSaison    // on lira la valeur HIVER ou PRINTEMPS ou ÉTÉ ou AUTOMNE
    si uneSaison = HIVER alors
        afficher "il neige"
    sinon
        si uneSaison = PRINTEMPS alors
            afficher "les fleurs poussent"
        sinon
            si uneSaison = ÉTÉ alors
```

Lien avec les entiers

Dans l'exemple ci dessus, on lit une Saison mais souvent, si on travaille avec les Mois par exemple, on disposera plutôt d'un entier. Il faut pouvoir convertir les valeurs. Chaque langage de programmation propose sa propre technique ; nous allons adopter la syntaxe suivante :

```
Saison(3)        // donne l'énumération de la saison numéro 3 (on commence à 1) ;
// donne ÉTÉ dans notre exemple.
position(uneSaison)        // donne l'entier associé à une saison ;
// si on a lu HIVER comme valeur pour uneSaison,
// donne la valeur 1.
```

Crédits

Ce document a été produit avec les outils suivants

- Les distributions `Ubuntu` et/ou `debian` du système d'exploitation `Linux`
- `LaTeX`/`Beamer` comme système d'édition
- `Git` et `GitHub` pour la gestion des versions et le suivi des corrections
- Les outils `make`, `rubber`, `pdfnup`, ...

Il est proposé sous licence