M1207 : Bases de la programmation

Emmanuel Hainry

IUT Nancy Brabois
Réseaux & Télécoms

2020

Première partie

Introduction

Définition

Algorithmique adj. et n. f. 1845 **1.** HIST. MATH. Relatif au système de numération dit algorithme. **2.** DIDACT. Science qui étudie l'application des algorithmes à l'informatique.

Algorithmique adj. et n. f. 1845 ~~**1.** HIST. MATH. Relatif au système de numération dit algorithme.~~ **2.** DIDACT. Science qui étudie l'application des algorithmes à l'informatique.

Algorithme n. m. 1554 latin médiév. Algoritmus latinisé de l'arabe Al khawarizmi (cf. algèbre) **1.** vx. Système de numération **2.** MATH. MOD. • suite finie et séquentielle de règles que l'on applique à un nombre fini de données permettant de résoudre des classes de problèmes semblables. • calcul, ensemble des actions nécessaires à l'accomplissement d'une tâche.

Objectif

- **Faire faire** quelque chose à l'ordinateur.
- **Réutiliser** un traîtement complexe.
- Résoudre un problème pour l'appliquer à **diverses données**.

- Similaire à une recette de cuisine.
- Similaire à un mode d'emploi (explication pas à pas).
- Similaire à un itinéraire.
- Similaire à la méthode de conversion décimal/binaire.

Analogie avec une recette

Recette de la tarte tatin
Ingrédients :
• 8 pommes • 1 pâte feuilletée • 2 sachets de sucre vanillé • un peu de cannelle • 100g de beurre • 100g de sucre en poudre
Pour obtenir une tarte tatin pour 6 personnes
Préparation :
Éplucher les pommes, les couper en 2 et enlever le cœur
Dans un moule à tarte rond, faire fondre le beurre directement sur le feu
Ajouter dans le plat le sucre et baisser le feu pour caraméliser
Disposer les pommes dans le plat
Saupoudrer de sucre vanillé et de cannelle
Recouvrir de pâte feuilletée et mettre au four pendant 35 minutes.

Un dégrèvement de la taxe d'habitation (appelé plafonnement) est accordé aux contribuables dont le revenu fiscal de référence n'excède pas les limites indiquées dans le tableau II ci-dessous et qui ne sont pas assujetis à l'ISF. Le revenu fiscal de référence tient compte aussi du revenu des occupants du logement s'il excède les limites indiquées dans le tableau I ci-dessous. Les occupants dont le revenu a été pris en compte pour le calcul du plafonnement sont indiqués dans le cadre "occupants".
Le dégrèvement est égal à la fraction de la cotisation brute de taxe d'habitation qui excède 3,44% du revenu fiscal de référence diminué d'un abattement (voir tableau III ci-dessous).
Le dégrèvement ainsi calculé est réduit d'un montant égal au produit de la base nette imposable la moins élevée par la différence entre le taux global de taxe d'habitation constaté dans la commune en 2000 et corrigé en 2011 pour prendre en compte le transfert d'une partie des frais de gestion. Le taux global correspond au cumul des taux votés par les collectivités bénéficiaires. Cette réduction n'est pas appliquée si elle est inférieure à 15€.
Une deuxième réduction du dégrèvement s'applique lorsqu'au moins une collectivité a supprimé ou diminué un des abattements de taxe d'habitation en vigueur en 2003. Cette réduction est égale à la différence positive entre le dégrèvement calculé comme indiqué ci-dessus sur la cotisation de l'année et le dégrèvement recalculé dans les mêmes conditions mais sur une cotisation déterminée en retenant les abattements de 2003. La cotisation dite "en référence 2003" est indiquée dans le tableau intitulé "Taxe d'habitation - Détail du calcul des cotisations".

Analogie avec un itinéraire

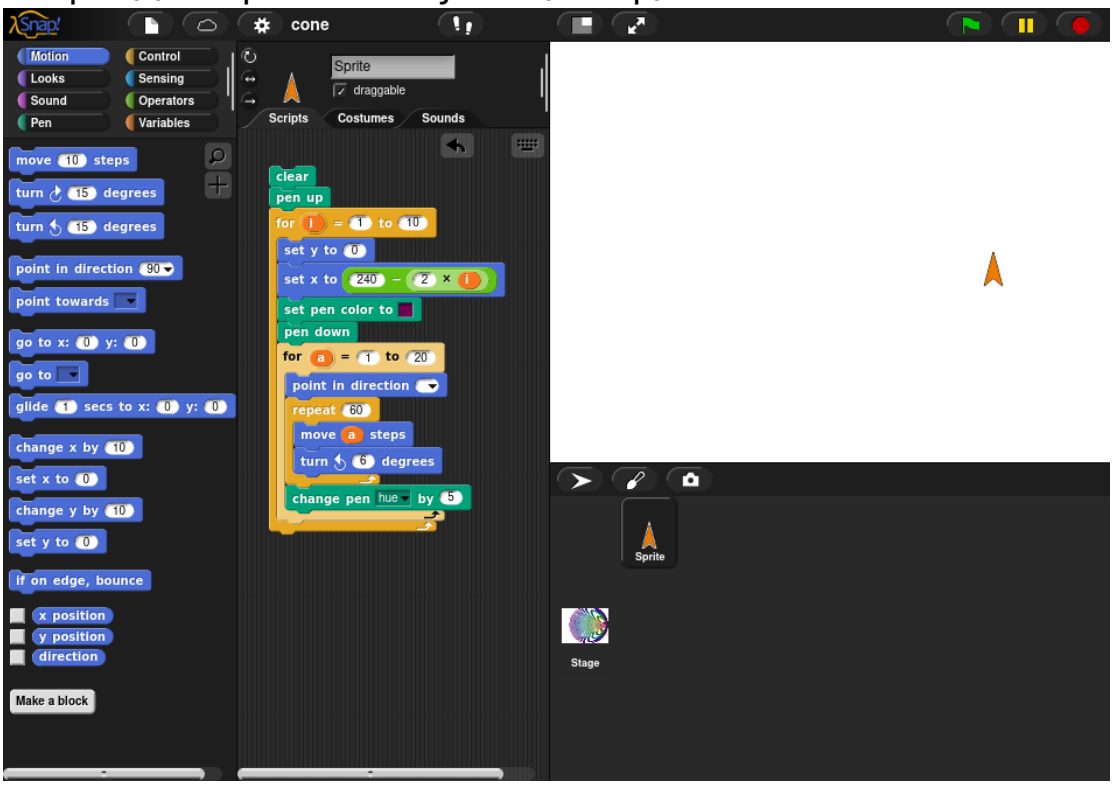
Du Loria à l'IUT
Ingrédients :
• Chaussures
Pour aller à l'IUT Nancy-Brabois.
Préparation :
Prendre la rue du jardin botanique vers le sud-ouest jusqu'à la rue de Vandœuvre.
Prendre à droite la rue de Vandœuvre.
Prendre à gauche la rue du doyen Urion.
Continuer sur 200 mètres.

Conversion décimal/binaire

Ingrédients :
• Nombre entier décimal n.
Préparation :
Si $n > 1$, effectuer la division de n par 2,
conserver le reste,
recommencer avec le quotient au lieu de n.
Ensuite, lire les restes du dernier au premier.

Dessin avec SNAP

`https://snap.berkeley.edu/snap/`

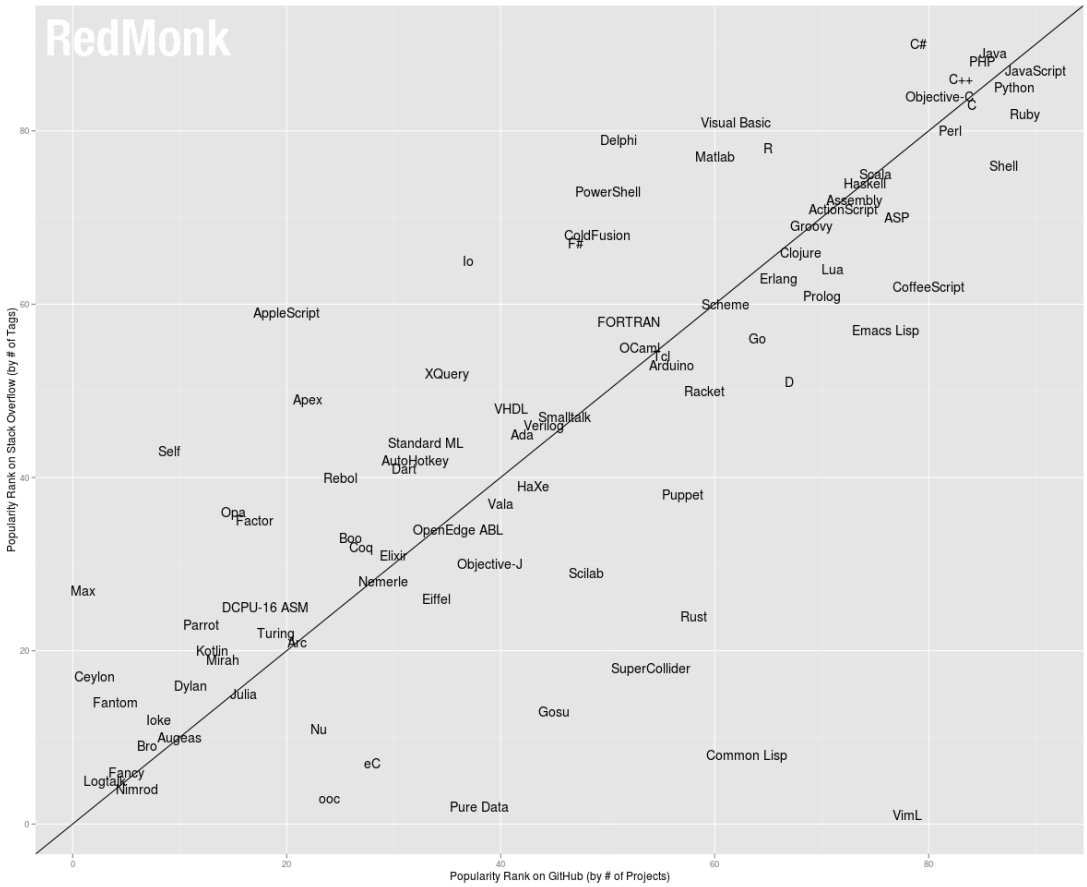


Points communs

Dans les différentes analogies, on retrouve des points communs :

- Un en-tête présentant ce que fait l'algorithme, ce qu'il manipule.
- Un corps qui détaille les étapes.
- Ces étapes peuvent être des actions (Éplucher, Appuyer, Continuer, Renvoyer) ; des séquences ordonnées (puis) ; des tests (Si un menu apparaît) ; des boucles (Mettre au four **pendant** 35 minutes, **Répéter**, **jusqu'à** la rue de Vandœuvre).

⇒ Possibilité de standardiser les algorithmes.



Source : `http://redmonk.com/sogrady/2012/09/12/language-rankings-9-12/`

Langages

- Nombreux langages différents.
- Différents paradigmes : impératif, objet, fonctionnel, aspect.
- Différentes conventions de typage : fort/faible, statique/dynamique, implicite/explicite.
- Différentes conventions de priorité, de composition...
- Diverses spécialisations : système, web, jeux, applications graphiques...
- L'algorithme est indépendant du paradigme.
- Il peut ensuite être traduit (implémenté) dans n'importe quel langage.

Algorithme vs Programme

- L'algorithme décrit la suite d'actions ou de calculs à effectuer pour effectuer la tâche.
  - Il s'écrit en *pseudo-code*.
  - Il doit être lisible et compréhensible par un humain (par exemple un prof d'informatique).
  - Il ressemble à un programme mais n'en est pas tout à fait un.
  - Il est suffisamment générique pour être traduit en un langage de programmation.
- Le programme traduit cette suite d'actions en langage d'ordinateur.
  - Le programme va être interprété par un ordinateur.
  - À ce titre, il doit respecter des contraintes de formes plus précises que l'algorithme.
  - Il peut être illisible par un humain (une preuve suit).

```
long long n,u,m,b;main(e,r)char **r;{f\
or(;n++||(e=getchar()|32)>=0;b="ynwtsflrabg"[n%=11]-e?b:b*8+
n)for(r=b%64-25;e<47&&b;b/=8)for(n=19;n;n["1+DIY/.K430x9\
G(kC["]-42&255^b||(m+=n>15?n:n>9?m%u*~-u:~(int)r?n+
!(int)r*16:n*16,b=0))u=1ll<<6177%n--*4;printf("%llx\n",m);}
```


Un algorithme, 4 traductions

Un même algorithme peut être traduit en de multiples langages. Pour illustrer cela, un algorithme de décomposition d'une somme en pièces et billets de 1, 2, 5 et 10 euros a été traduit en C, en Python, en Lua et en Haskell.

```c
#define loop(x,m,M) for(int x=0; x<=M; ++x)
#include <stdio.h>

// Algorithme de décomposition d'une somme en pièces

void main()
{// Début
        int somme;
        scanf("%d", &somme);
        int reste;
        int p10, p5, p2, p1;
        reste = somme;
        p10 = 0;
        p5 = 0;
        p2 = 0;
        p1 = 0;

        while (reste >= 10)
        {
                reste = reste - 10;
                p10 = p10 + 1;
        }
        while (reste >= 5)
        {
                reste = reste - 5;
                p5 = p5 + 1;
        }
        while (reste >= 2)
        {
                reste = reste - 2;
                p2 = p2 + 1;
        }
        p1 = reste;

        printf("%d pièces de 10", p10);
```



```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
"""Algorithme de decomposition d'une somme en pieces"""

# Entree
#        somme: entier
somme = int(raw_input("Somme a decomposer: "))
# Sortie
#        -
# Variables
#        reste: entier
#        p1, p2, p5, p10: entier

# Debut
p10 = 0
p5 = 0
p2 = 0
p1 = 0
reste = somme
while reste >= 10:
        p10 = p10 + 1
        reste = reste - 10
while reste >= 5:
        p5 = p5 + 1
        reste = reste - 5
while reste >= 2:
        p2 = p2 + 1
        reste = reste - 2
p1 = reste
```

```lua
#!/usr/bin/lua
-- Algorithme de décomposition d'une somme en pièces

-- Entrée : somme: entier
somme = tonumber(io.read())
-- Sortie : -
-- Variables : reste: entier
--             p1, p2, p5, p10: entier

-- Début
p10, p5, p2, p1 = 0, 0, 0, 0
reste = somme
while reste >= 10 do
        p10 = p10 + 1
        reste = reste - 10
end
while reste >= 5 do
        p5 = p5 + 1
        reste = reste - 5
end
while reste >= 2 do
        p2 = p2 + 1
        reste = reste - 2
end
p1 = reste
print(p10, "pièces de 10,", p5, "pièces de 5,")
print(p2, "pièces de 2,", p1, "pièces de 1.")
```



```haskell
-- Algorithme de décomposition d'une somme en pièces

decompose s = decompose_aux s 0 0 0 0

decompose_aux s p1 p2 p5 p10
        | s>=10 = decompose_aux (s-10) p1 p2 p5 (p10+1)
        | s>=5 = decompose_aux (s-5) p1 p2 (p5+1) p10
        | s>=2 = decompose_aux (s-2) p1 (p2+1) p5 p10
        | otherwise = (s, p2, p5, p10)

ch (z,b,c,d) = show d ++ " pièces de 10\n" ++ (show c) ++ " pièces de 5\

main = do
        somme <- getLine
        putStr (ch (decompose (read somme :: Int)))
```

Deuxième partie

Syntaxe d'un algorithme

Notre langage algorithmique

Formalisme

Un algorithme représente la *suite de règles* à appliquer pour résoudre un problème.
Il doit être *lisible* et *compréhensible p*ar plusieurs personnes
Pour cela, il devra suivre des règles et être indépendant d'un langage de programmation.
Nous utiliserons pour décrire nos algorithmes une syntaxe inspirée de celle de *lua* [1]

1. Et nous pourrons donc les tester avec l'interpréteur `lua`.

Dialogue avec l'utilisateur

- Afficher du texte : `print("Blabla")`
- Obtenir une réponse : `io.read()`

Exemple `print("Bonjour")`

- `reponse = io.read()`
- `print(reponse)`

Interpréteur

L'interpréteur permet de tester les instructions au fur et à mesure.

- Démonstration

```
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> print("Hello world")
Hello World
> print("Bonjour")
Bonjour
> print(7)
7
> print(4+3)
7
```

Notre premier programme

Affichage d'un message. Stockage dans une variable, réutilisation, concaténation. Séquence. Test : la structure `if ... then ... else ... end`. Expression booléenne "complexe". Répéter une suite d'actions 3 fois. Répéter une suite d'actions jusqu'à un événement. Une variable pour calculer le score.

```
print("Comment vous appelez-vous ?")
nom = io.read()
print("Bonjour, " .. nom)
score = 0
for i = 1, 3 dorepeat
     print("Quelle est la capitale de la Belgique ?")
     reponse = io.read()
     if reponse == "Bruxelles" thenor reponse == "Brussel" then
               print("Bravo")
               score = score + 1
     else
               print("Perdu")
     end
end  print("Continuer ? [o/n]")
     continue = io.read())
```


Notre premier programme complet

Résumé

- `print` et `io.read`.
- `var = valeur`
- `..` pour joindre (concaténer) deux chaînes.
- `if cond then ... else ... end` pour effectuer des actions conditionnelles.
- `for i = 1, 17 do ... end` pour répéter 17 fois une action.
- `repeat ... until cond` pour repéter jusqu'à un événement.
- `while cond do ... end` pour répéter tant qu'une condition est vérifiée.
- `t = { 1, 2, 12, -56}` et `t[3]` pour les tableaux.

Troisième partie

Formalisme, variables et types

Composants d'un programme

- Variables : 

- Expressions : 

- Instructions : 

- Boucles : 

Formalisme

Un algorithme se compose

- d'un en-tête qui spécifie
  - le **nom** de l'algorithme,
  - le **rôle**, c'est-à-dire l'utilité de l'algorithme
  - les données en **entrée**, c'est-à-dire sur lesquelles il travaillera (assorties éventuellement de préconditions, c'est-à-dire de conditions que doivent remplir ces données)
  - les données en **sortie**, c'est-à-dire ce qui sera produit par l'algorithme (assorties éventuellement de postconditions, c'est-à-dire de conditionsque rempliront les données à la fin de l'algorithme)
  - les données locales de travail (**déclarations**).
- d'un corps qui se compose
  - du mot clé **début**
  - d'une suite d'*instructions indentées*
  - du mot clé fin

Exemple

```
Nom : AddDeuxEntiers
Rôle : Addition de deux entiers
Entrée : a,b : entiers
Sortie : c : entier
Déclaration : -

Début
  c = a+b
Fin
```


Qu'est-ce qu'une variable

Une variable est une entité contenant une information.

- Une variable possède un nom : son **identifiant**.
- Une variable possède une **valeur**.
- Une variable possède un **type** caractérisant l'ensemble des valeurs qu'elle peut prendre.

Les variables sont stockées dans la mémoire de l'ordinateur.

Analogie

On peut faire l'analogie entre les variables et des archives dans une armoire :

- l'armoire représente la mémoire de l'ordinateur
- les tiroirs représentent les variables
- l'étiquette du tiroir correspond à l'identifiant de la variable
- le contenu correspond à la valeur de la variable
- la forme et la taille du tiroir correspondent au type (un tiroir à factures, un tiroir à bons de commandes...)

Manipuler les variables

Certaines variables ont un rôle particulier :

- Les variables d'entrées
- Les variables de sortie

Au début de l'algorithme, les variables qui ne sont pas d'entrée n'ont pas encore de valeur, elles seront (éventuellement) initialisées plus tard.
L'essentiel des algorithmes consistera à manipuler les variables :

- faire des calculs sur les variables
- stocker les résultats de calculs
- agir en fonction de la valeur d'une variable
- …

Actions sur les variables

On ne peut faire que deux choses avec une variable :

- Obtenir sa valeur (regarder son contenu)
  - Cela s'effectue en nommant la variable

Exemple `toto`

Note Certains langages de programmation utilisent un caractère spécial pour différencier la valeur et l'identifiant. Par exemple en shell, `$toto`.

- Affecter une nouvelle valeur.
  - Cela s'effectue à l'aide de l'opérateur d'affectation : =.

Exemple `toto = 7`

Note Certains langages de programmation utilisent d'autres symboles ou des mots clefs, mettant en valeur le caractère directionnel de l'affectation.

Pascal `toto := 7`
Haskell `toto <- 7`
Csh `set toto = 7`
Tcl `set toto 7`
Lisp `(setq toto 7)`
Snap 

Types de données

Le type d'une variable caractérise

- l'ensemble des valeurs que peut prendre cette variable,
- l'ensemble des actions que l'on peut effectuer sur cette variable.

Dans l'entête de l'algorithme, on déclare les types des variables utilisées (que ce soit en entrée, sortie ou déclaration).

Syntaxe `identifiant : type`

Par exemple

- `age : naturel`
- `nom : chaîne`

Préconditions

En plus du type, on peut ajouter des préconditions sur les variables d'entrée.

Syntaxe `assert(condition)`

Par exemple

- `a : réel`
- `assert(a>17)`

Types de données

Le type d'une variable est déclaré une fois pour toute (une variable ne peut pas changer de type) [2].
La valeur d'une variable doit obligatoirement être de ce type (on ne peut pas faire rentrer de facture dans le classeur à bons de commandes).
Dans l'exemple précédent, `a` et `b` sont déclarés comme des entiers

- `a` et `b` ne pourront pas contenir des réels !
- on ne peut pas décider de nommer `a` une nouvelle variable sensée contenir autre chose.

On distingue les *types simples* et les *types complexes*.

2. Cette contrainte est souvent appelée *typage fort*. Il s'agit d'une convention que ne respectent pas tous les langages : java la respecte mais pas lua.

Types simples

Les types simples permettent de stocker les nombres, les booléens, les chaînes de caractères...
Il y a deux grandes familles de types simples :

- les types finis
- les types simples infinis

Types simples finis

booléen	les variables ne peuvent prendre que les valeurs `true` et `false`[3].
intervalle	les variables ne peuvent prendre que des valeurs entières définies dans l'intervalle, par exemple $[1..10]$
énuméré	les variables ne peuvent prendre que des valeurs explicitées.
	▶ exemple : les jours de la semaine.
	▶ Ce sont les seuls types simples définissables pas le programmeur.
caractère	les variables ne peuvent prendre que des valeurs de la forme `'à'`, `'λ'`.

- Exemples :
  - `masculin : booléen`
  - `mois : [1..12]`
  - `jour : JoursDeLaSemaine`

3. 0 et 1 ne sont pas des booléens, pas plus que V et F, oui et non.

Types énumérés

Les types énumérés doivent être définis explicitement dans l'entête de l'algorithme en énumérant les valeurs possibles.

Syntaxe `nomDuType = {valeur1, valeur2, ..., valeurN}`

Exemple `JoursDeLaSemaine = {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche}`

Types simples infinis

Les types simples infinis peuvent prendre des valeurs en nombre infini, éventuellement indénombrable.

naturel Les entiers positifs, par exemple, 0, 7, 2147483648.

entier Les entiers positifs ou négatifs, par exemple 0, -7, 2147483648.

réel Les nombres réels, par exemple 0, 2.3, `math.pi`.

chaîne Les chaînes de caractères. Par exemple `"Algorithmique"` ou "$K\alpha\lambda\eta\mu\epsilon\ \kappa o\sigma\mu\epsilon$"

- Exemples
  - `age : naturel`
  - `taille : réel`
  - `nom : chaîne`
- Représentation des types numériques

Digression : représentation des réels

Les nombres réels ont besoin d'être représentés en binaire pour être utilisés

lua Les nombres sont stockés sur 64 bits (double précision).

java Les réels peuvent être *double* (64 bits) ou *float* (32bits).

C Les réels peuvent être *double* (64 bits), *float* (32 bits) ou *short* (16 bits).

Dans tous les cas, réels stockés suivant la norme IEEE 754.

Exemple Pour un double, il y a

- 1 bit de signe
- 11 bits d'exposant
- 52 bits de mantisse

Le double $a_1 a_2 ... a_{63} a_{64}$ représente le nombre réel

$$(-1)^{a_1} \times m \times 2^e$$

où

$$e = \sum_{i=2}^{12} 2^{12-i} a_i - 1023$$

$$m = 1 + \sum_{i=13}^{64} 2^{12-i} a_i$$

Types complexes

Il existe et il est possible de définir des types complexes, c'est-à-dire des types pouvant contenir plusieurs attributs (variables internes) et des capacités propres.
Cette notion sera vue en profondeur en M2207 (Programmation Objet).

Dans ce module, les seuls types complexes étudiés seront les tableaux

Tableaux

Les tableaux sont des structures dans lesquelles on peut mettre plusieurs éléments de même type.

- On peut créer des tableaux de chaînes de caractères, des tableaux de booléens, des tableaux de tableaux d'entiers...
- Un tableau doit être déclaré : `identifiant : tableau de type`
- Les cases sont numérotées de 1 à la longueur du tableau.
- Chaque case peut être imaginée comme une variable dont l'identifiant est `identifiant[n]` où `n` est le numéro de la case.
- Les tableaux sont de la forme `{e1, e2, e3}` où e1, e2... sont des éléments du type correspondant.

Exemple `t : tableau d'entiers`

Exemple `{false, true, false}` est un tableau de booléens.

Exemple `{{1,2}, {2,3}, {0,0}}` est un tableau de tableaux d'entiers.

Quatrième partie

Expressions et instructions

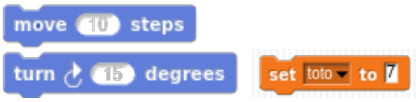
Expressions
- Définitions
- Conventions
- Évaluation des expressions complexes
- Opérateurs spécifiques aux types simples

Instructions
- Affectations
- Entrées/Sorties
- Séquentialité

Expressions et instructions

Un algorithme est une suite d'instructions.

Les *instructions* représentent les actions dans l'algorithme.
Le corps de l'algorithme peut n'être composé que d'une instruction (exemple : addDeuxEntiers).



Les *expressions* sont des descriptions de "calculs".
Elles ont une valeur mais n'agissent pas dessus.
Elles ne peuvent pas être isolées en dehors d'une instruction.

Expressions

- Une expression représente un calcul.
- Une expression est une combinaison d'opérateurs et d'opérandes.
- Un *opérateur* est un symbole d'opération (par exemple $+$).
- Un *opérande* est une entité utilisée par un opérateur. Il peut s'agir d'une variable, d'une constante ou d'une expression.
- Une expression possède une *valeur* (que l'on peut calculer, évaluer) et un *type* (que l'on peut inférer).
- La valeur sera calculée lors de l'exécution de l'algorithme. Elle est donc dynamique, dépend des conditions initiales.
- Le type est en général inféré de façon statique à la compilation et ne change pas d'une exécution à l'autre.

Expressions

Les expressions sont de la forme

$$v$$

où v est une constante ou une variable,

$$e_1 \; op \; e_2$$

où e_1 et e_2 sont des expressions et op un opérateur binaire,

$$op \; e$$

où e est une expression et op un opérateur unaire,

$$f(e)$$

où f est une fonction et e une expression.

Expressions, opérateurs et opérandes

Par exemple, 5, a, `"Bonjour"`, a+7 ou a+b sont des expressions.

Dans `a+b`,

- `a` et `b` sont les opérandes. `a` est l'opérande gauche, `b` est l'opérande droit.
- `+` est l'opérateur.
- `a+b` est une expression utilisable dans une autre expression ou dans une instruction.

Si `a` vaut 2 et `b` vaut 3, alors l'expression `a+b` vaudra 5.
Si `a` et `b` sont de type *entier*, alors, `a+b` sera également de type entier.

Opérateurs

Un opérateur peut être unaire ou binaire

- unaire s'il n'admet qu'un seul opérande. Par exemple, le `non` logique ;
- binaire s'il admet deux opérandes. Par exemple, l'opérateur +.

- Un opérateur est associé à un type de données. Il ne peut être utilisé qu'avec des données de ce type.
- Par exemple, l'opérateur `non` ne peut être utilisé qu'avec des expressions booléennes.
- Par exemple, l'opérateur + peut être utilisé avec deux opérandes entiers, deux opérandes naturels, deux opérandes réels.
- Mais on ne peut utiliser + avec une chaîne et un entier.

Opérateurs et types

- $+$, $-$, $\times$... sont des opérateurs binaires
- round, sqrt sont des opérateurs unaires
- les types sont représentés par des formes :
  - les rectangles devraient être ovales...
  - les ovales sont des nombres
  - les hexagones sont des booléens

Opérateurs : attention

> *"The problem with putting two and two together is that sometimes you get four, and sometimes you get twenty-two."*
>
> Dashiell Hammett

! On ne peut pas *a priori* additionner un entier et un réel.

- Dans certains cas, on peut utiliser un opérateur avec des opérandes de types différents.
- La signification d'un opérateur peut changer en fonction du type des opérandes :
  - `+` pour les entiers représente l'addition.
  - Dans de nombreux langages (java, C, python), `+` pour les chaînes représente la concaténation.
  - `2+3` vaut `5`.
  - En python et java, `"2" + "3"` vaut `"23"`
  - En lua, `"2" + "3"` vaut 5 (et c'est un entier)!
  - En lua, `"2" .. "3"` vaut `"23"`.
  - En python, `"2" * 3` vaut `"222"`.
  - En lua, `"2" * 3` vaut `6`.
  - En java, `"2" * 3` ne veut rien dire.

Opérateurs : conventions de typage

- Dans certains cas, on peut utiliser un opérateur avec des opérandes de type différents.

Par exemple, il est envisageable d'écrire `2 + 3.5`.

- Différentes conventions quant au typage de telles expressions sont envisageables.
  1. conversion de chaque opérande au plus grand type.
  2. conversion de l'opérande droit au type de l'opérande gauche.
  3. conversion du résultat au plus petit type pouvant le contenir.
- Dans ces conditions,
  1. 2 est converti en réel. L'expression vaut `5.5` et est de type réel.
  2. `3.5` est converti en naturel. L'expression vaut `5` et est de type naturel.
  3. Le résultat est le même que dans les cas précédents.

Note On peut stocker une valeur entière dans une variable réelle, mais pas l'inverse. Rien n'indique que 2 n'est pas un réel...

Exercice Considérons l'expression `2 * 3.5`. Quels sont sa valeur et son type dans les diverses conventions proposées ?

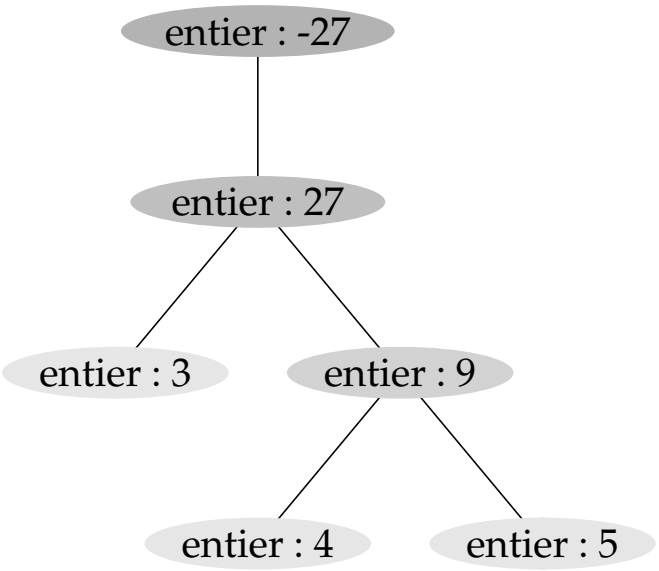
Priorité entre opérateurs

Nous avons vu que les opérandes peuvent être des expressions.
Par exemple `3 * a + 3` est une expression.

- Nécessité de fixer des priorités entre les opérateurs.
- Utiliser des parenthèses pour éviter toute ambiguïté.

Expressions : arbre d'évaluation

Considérons l'expression `-(3*(a+b))`. Pour l'évaluer, nous allons réaliser son arbre d'évaluation. En supposant a et b entiers ayant pour valeurs respectives 4 et 5.



- On annote les feuilles en mettant leur type et leur valeur.
- On calcule les valeurs des sous-arbres et les types en remontant dans l'arbre jusqu'à la racine.

Opérateurs booléens

Les opérateurs booléens sont les opérateurs logiques classiques : non, ou et et.

not opérateur unaire : non logique

or opérateur binaire : ou logique

and opérateur binaire : et logique

a	not a
false	true
true	false

a	b	a or b
false	false	false
false	true	true
true	true	true
true	false	true

a	b	a and b
false	false	false
false	true	false
true	true	true
true	false	false

Opérateurs sur les énumérés

ord Index (numéro de l'énuméré spécifié dans la bijection énuméré → naturel).

pred Prédécesseur (énuméré précédent, celui d'index juste inférieur).

succ Successeur (énuméré suivant, celui d'index+1).

Exemples

- `succ(Lundi)` vaut `Mardi`
- `succ(Dimanche)` vaut `Lundi`
- `pred(Mardi)` vaut `Lundi`
- `pred(Lundi)` vaut `Dimanche`
- `ord(Lundi)` vaut 1
- `ord(Dimanche)` vaut 7

Opérateurs sur les caractères

- Opérateurs sur les types énumérés

char Opérateur associant un caractère à son code ASCII

Exemple `ord(A)` vaut 65

Exemple `char(65)` vaut `A`

Exemple `pred(A)` vaut `@`

Opérateurs sur les types numériques

- Opérations usuelles : `+`, `-`, `*`, `/` et `%`.
- `+`, `-` et `*` fonctionnent avec deux opérandes de même type. Le résultat est de ce même type.
- `%` (modulo) ne fonctionne qu'avec des opérandes naturels.
- Il est possible de calculer avec des nombres de types différents, mais il faut choisir une convention de typage.
- La division fonctionne avec des opérandes de n'importe quel type. Il s'agit de la division réelle [4].
- Que donne la division par 0 ?

En lua	En haskell	En python	En java
`> =2/0` `inf`	`ghci> 2/0` `Infinity`	`> 2/0` `ZeroDivisionError`	`groovy> 2/0` `java.lang.ArithmeticExcep`
`> =0/0` `nan`	`ghci> 0/0` `NaN`	`> 0/0` `ZeroDivisionError`	`groovy> 0/0` `java.lang.ArithmeticExcep`

4. Ce ne sera pas le cas en java.

Opérateurs pour les chaînes de caractères

Concaténation La concaténation est représentée par `..`

Longueur `# ch` donne la longueur de `ch`.

Recherche `string.find(chaine, motif)` permet de rechercher un motif dans une chaîne.

Remplacer `string.gsub(chaine, motif, rep)` permet de remplacer toutes les occurences de motif dans chaine par rep.

Sous-chaine `string.sub(chaine, debut, fin)` donne la sous-chaîne de `debut` à `fin`

Exemple `string.find("Bonjour à tous", "ou")` vaut (5, 6).

Exemple `string.gsub("Bonjour " .. "à tous", "ou", "a")` vaut `"Bonjar à tas"`.

Exemple `# "Bonjour à tous"` vaut 15.

Exemple `string.sub("Bonjour à tous", 5, 12)` vaut `"our à t"`.

Opérateurs sur les tableaux

taille `#` opérateur unaire qui donne la taille du tableau

insertion `table.insert(identifiant, valeur)`

Demo `t = {false, true, false}`

Demo `table.insert(t, false)`

Demo `= #t`

Demo `= t[2], t[4]`

Demo `= t[#t]`

Opérateurs de comparaison

Pour les types simples, il est possible de savoir si deux entités ont la même valeur.

- L'opérateur d'égalité ==.
- Permet de comparer deux opérandes de même type.
- Expression booléenne.

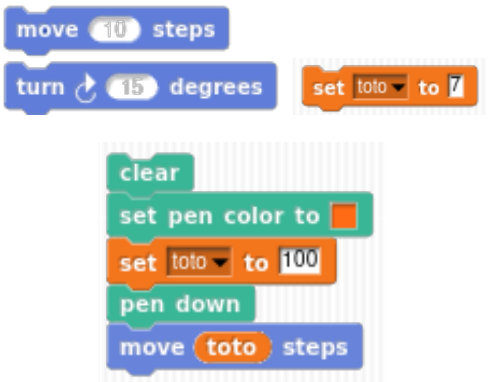
Autres opérateurs :

- Non-égalité : $\sim=$
- Comparaison : <, <=, >=, > (pour les types comparables : les nombres, les caractères).

Pour les types complexes, cela ne teste pas l'égalité mais l'identité.

Instructions

- Les instructions sont les actions dans l'algorithme.
- Affectations et effets de bord (lectures/écritures de toutes sortes)
- Au moins une instruction par algorithme.
- Les instructions sont des statements.
- La séquence de 2 statements est un statement.



Affectation

L'affectation assigne une valeur à une variable.
La syntaxe en est : `identifiant = expression` où `expression` est une expression dont le type est compatible avec celui de la variable `identifiant`. La valeur stockée est celle de l'expression.

écomposition Prenons l'instruction `c = a + b`.

1. On évalue l'expression à droite du =.
2. Pour cela, on obtient les valeurs des variables `a` et `b` et on les additionne.
3. Le résultat de cette addition est stocké dans la variable `c`.
4. Si `c` possédait une valeur avant l'affectation, cette valeur est perdue.

Entrées/Sorties

Un algorithme peut avoir des interactions avec l'utilisateur.
Il peut par exemple afficher un résultat (du texte, le contenu d'une variable).
Il peut demander à l'utilisateur de saisir une information (donnée) qui sera stockée dans une variable.
Pour celà, nous allons utiliser des instructions pour **écrire** et **lire**. [5]

5. Ces termes sont à comprendre du point de vue de la machine : celle-ci va écrire ou afficher un résultat et obtenir une donnée en la lisant.

Entrées/Sorties

Nous utiliserons en algorithmique la nomenclature de lua (qui est assez standard) :

- `print(expression)` va afficher la valeur de l'expression.
- `print(a1, a2, a3)` va afficher les valeurs de a1, a2 et a3 séparées par des tabulations.
- `a = io.read()` va attendre une valeur de l'utilisateur puis la stocker dans la variable `a`.

Demo
```
print("Bonjour")
```
Demo
```
print(2+3)
```
Demo
```
print("2+3")
```
Demo
```
a = io.read()
print("a vaut " .. a)
```
Demo
```
a = io.read()
print("a :", a)
```


Entrées/Sorties et tubes

Vu en M1105 : redirections, tubes.

Comparons

```
$ lua decomposition.lua
$ echo 12 | lua decomposition.lua
$ lua decarg.lua 12
```

Exemples

euroVersFranc.lua euroVersFranc.lua

```
lua euroversfranc.lua 33
```

---

euroVersFranc2.lua

```
echo 33 | lua euroversfranc2.lua
```


Séquentialité

Quand nous écrivons plusieurs instructions séparées par des passages à la ligne, elles sont exécutées séquentiellement, dans l'ordre.

Pas de possiblité d'effectuer d'actions simultanées dans ce formalisme.

Trace d'exécution

Pour étudier un algorithme (comprendre ce qu'il fait, vérifier qu'il fonctionne...), on a recours à l'outil de la *trace d'exécution*.

- On décrit les valeurs des variables après l'exécution de chaque instruction.
- On réalise un tableau dont les colonnes décrivent les variables, les lignes les instructions.
- Si une variable n'a pas encore de valeur, on ne met rien dans la case correspondante.
- Des lignes spéciales correspondant au début et à la fin de l'algorithme.
- Au début, seules les variables d'entrée ont une valeur (choisie).
- À la fin, seules les variables de sortie ont une valeur.

Exemple pour trace d'exécution

```
Nom : AddDeuxEntiers
Rôle : Addition de deux entiers
Entrée : a,b : entiers
Sortie : c : entier
Déclaration : d : entier

Début
  c = a+b
  a = c+a
  b = (a-b)*5/2
  d = a/(b-c)
Fin
```

Trace d'exécution : exemple

	a	b	c	d	
Début	7	3			
`c = a+b`	7	3	10		
`a = c+a`	17	3	10		
`b = (a-b)*5/2`	17	35	10		
`d = a/(b-c)`	17	35	10	0,68	
Fin			10		

Trace d'exécution : exemple

	a	b	c	d	
Début	16	64			
`c = a+b`					
`a = c+a`					
`b = (a-b)*5/2`					
`d = a/(b-c)`					
Fin					

Cinquième partie

Structures de contrôles

Ordinogramme

Conditionnelles

Boucles

Structures de contrôle

Pour l'instant, tout est séquentiel : une ligne = une action exécutée une fois.
Nous allons par la suite vouloir effectuer des actions uniquement dans certains cas.
Nous allons vouloir répéter des actions.
Pour cela, nous allons ajouter des structures de contrôle permettant des choix (conditionnelles) et des répétitions (boucles).
Les structures sont (comme les instructions) des statements.
Mais avant, nous allons voir un outil permettant de représenter le flot de contrôle d'un algorithme : *l'ordinogramme*.

Ordinogramme

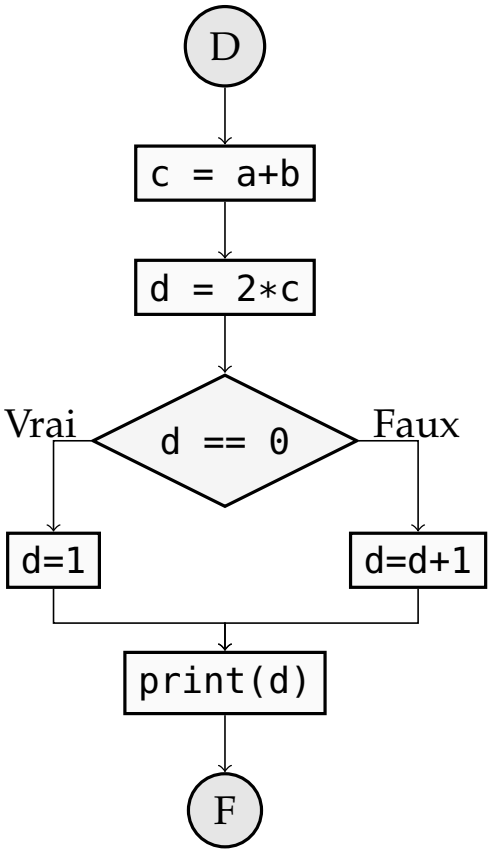
Un ordinogramme est un graphe décrivant le flot de contrôle, c'est-à-dire la suite des instructions qui vont être réalisées et les choix qui vont être faits.
Un ordinogramme comporte trois sortes de nœuds :

- les nœuds d'instructions représentés par des rectangles contenant l'instruction ;
- les nœuds de choix représentés par des losanges contenant une expression booléenne ;
- les nœuds de début et de fin, représentés par des disques.

Les nœuds sont reliés par des flèches :

- le nœud début possède uniquement une flèche sortante ;
- le nœud fin possède uniquement une flèche entrante ;
- les nœuds d'instruction possède une flèche entrante et une flèche sortante ;
- les nœuds de choix possèdent une flèche entrante et deux flèches sortantes.

Exemple d'ordinogramme



Conditionnelles

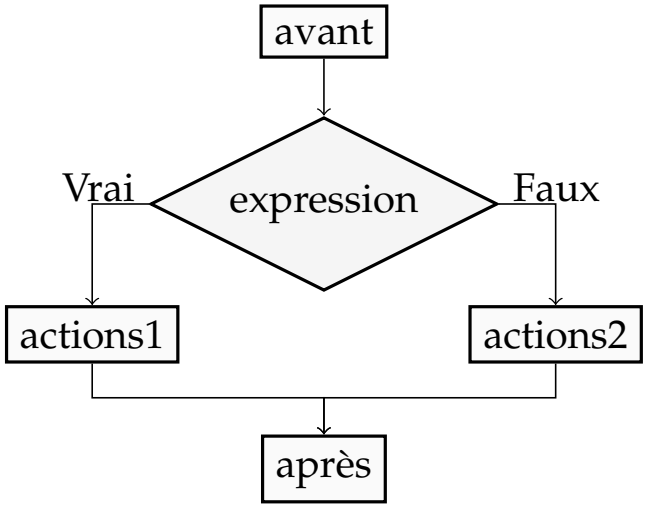
Les conditionnelles permettent de réaliser des actions conditionnelles, de faire des choix.

En français, **Si** une condition est réalisée... Ou **Au cas où** une variable vaut une valeur donnée.

En anglais, **if** et **case**

Conditionnelle simple

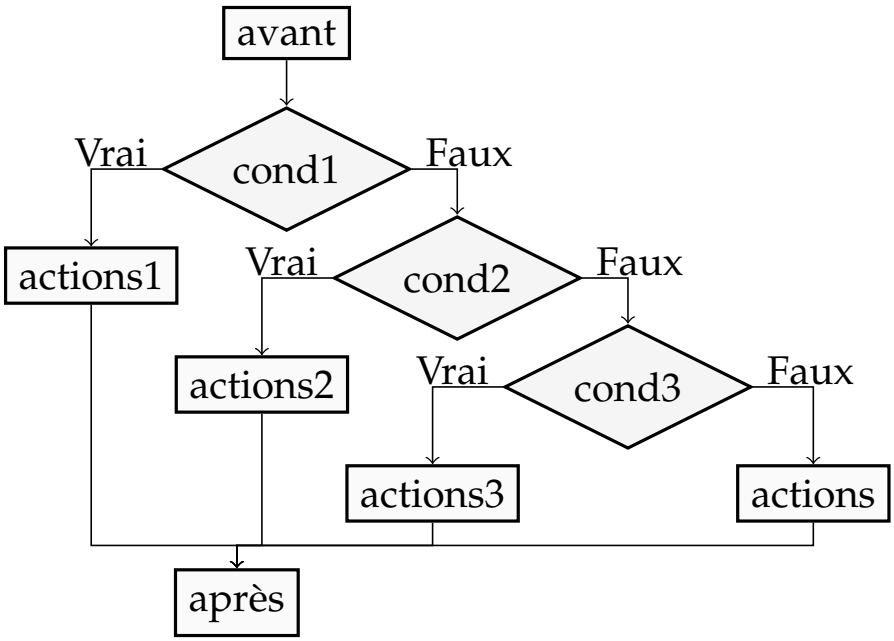
```
avant
if expression then
        actions1
else
        actions2
end
après
```



- expression est une expression booléenne.
- actions1 et actions2 sont des suites de statements.
- actions1 et actions2 peuvent contenir des conditionnelles.
- La partie `else` est optionnelle.

Conditionnelle multiple

```
avant
if cond1 then
        actions1
elseif cond2 then
        actions2
elseif cond3 then
        actions3
else
        actions
end
après
```



Aiguillages

Les aiguillages (cas) sont des conditionnelles multiples où l'on teste la valeur d'une expression donnée.

```
case expression
when val1
        act1
when val2
        act2
when val3
        act3
else
        act
end
```

```
if expression == val1 then
        act1
elseif expression == val2 then
        act2
elseif expression == val3 then
        act3
else
        act
end
```

```
Nom : moisde30jours
Rôle : Déterminer si un mois comporte 30 jours
Entrée : mois : entier
Sortie : resultat : booléen
Déclaration : -

Début
  case mois
  when 4, 6, 9, 11
        resultat = true
  when 2
        resultat = false
  else
        resultat = false
  end
Fin
```


Itérations

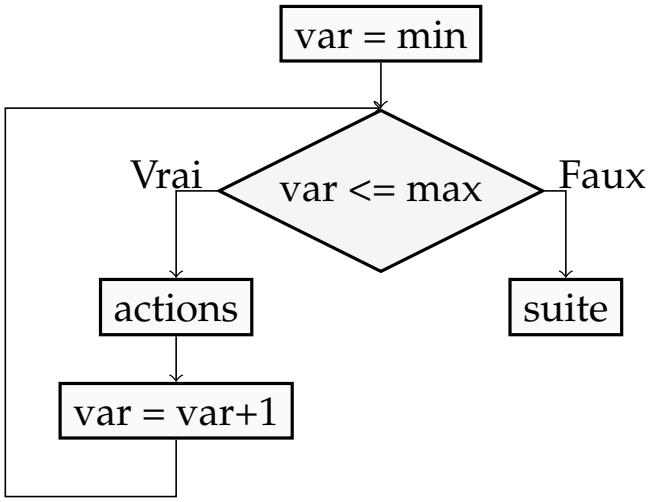
Pour répéter un certain nombre de fois un traitement, on utilisera des itérations.

Il existe deux sortes d'itérations : les itérations bornées (on connaît dès le début de la boucle le nombre d'itérations à faire) et les itérations non bornées (on ne sait pas combien de fois la boucle sera parcourue).

Cela se traduit en français par des mots clefs comme **répéter n fois**, **répéter jusqu'à** et **tant que**. En algorithmique, on utilisera **for ... do**, **while ... do** et **repeat ... until**.

Itération bornée

```
for var = min, max do
        actions
end
suite
```



- `min` et `max` sont des expressions entières.
- actions est une suite de statements.
- actions va être exécuté $max - min + 1$ fois.
- On peut utiliser `var` dans actions comme une variable normale.

Demo `for i = 1, 3 do print(i) end`


```
-- Nom : Somme
-- Rôle : Calculer la somme des n premiers entiers positifs
-- Entrée : n : naturel
-- Sortie : s : naturel
-- Déclaration : i : naturel

-- Début
s = 0
for i = 0, n-1 do
    s = s+i
end
-- Fin
```

Exercice Ordinogramme de cet algorithme.

Exercice Trace d'exécution pour $n = 3$.

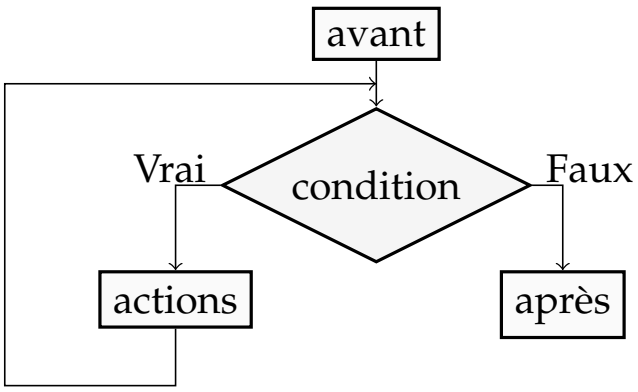
Itération non bornée

Dans le cas de l'itération bornée, on précise le nombre d'itérations qui auront lieu dans la définition de la boucle : $max - min + 1$.
Dans certains cas, on ne sait pas prévoir combien de fois il faudra parcourir la boucle, mais on sait donner une condition pour ne plus la parcourir.

- L'intérêt est de conditionner la répétition d'un traîtement, cela signifie que l'on passe à l'itération suivante sous réserve d'une condition.
- Deux constructions vont exister :
  - Répéter jusqu'à ce que
  - Tant que faire

Boucle tant que

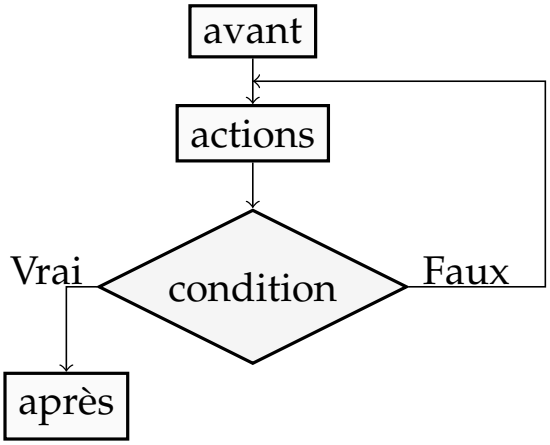
```
avant
while condition do
        actions
end
après
```



- Attention, cette boucle peut ne pas terminer si la condition ne devient jamais fausse !
- Il faut donc dans actions penser à modifier ce qui constitue condition.

Boucle répéter jusqu'à

```
avant
repeat
      actions
until condition
après
```



- Attention, cette boucle peut ne pas terminer si la condition ne devient jamais fausse !
- Contrairement à la boucle tant que, actions est nécessairement exécutée au moins une fois.

Exercice

- Au vu des ordinogrammes, comment peut-on simuler une boucle repeat à l'aide d'une boucle while ?
- Comment simuler une boucle while à l'aide d'une boucle repeat ?

Exercice Ordinogramme de cet algorithme

Exercice Trace d'exécution pour $n = 9$ puis pour $n = 12$.

Exercice

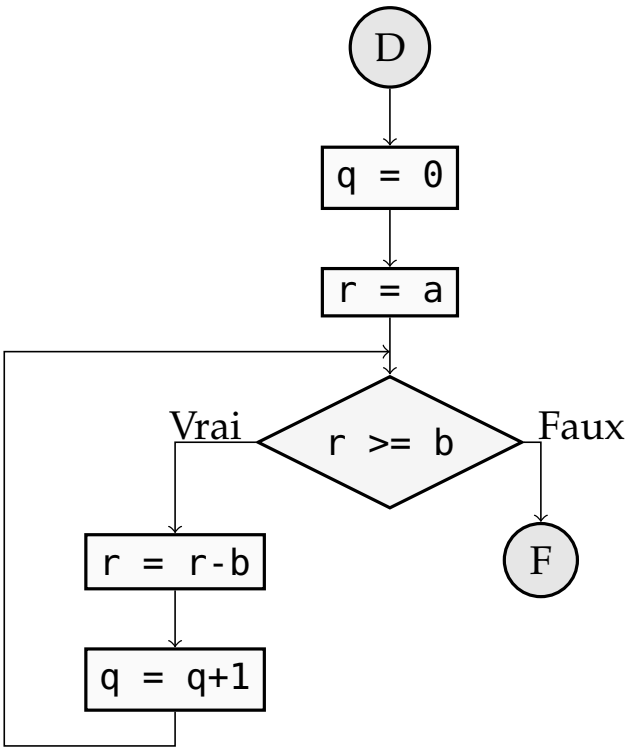
Réaliser et tester un algorithme calculant le quotient et le reste dans une division euclidienne.

- Étant donnés a et b, on veut *quotient* et *reste* tels que $0 \leq reste < b$ et $a = quotient \times b + reste$.
- L'idée sera de vérifier que l'on peut faire la division et si c'est le cas de faire diminuer *reste* en incrémentant *quotient* de façon à ce que l'équation $a = quotient \times b + reste$ soit toujours vérifiée.

- Dessiner l'ordinogramme de l'algorithme obtenu.
- Faire des traces d'exécutions pour des valeurs intéressantes (par exemple $(13, 0)$, $(17, 5)$, et $(12, 4)$).

Correction : algorithme

Correction : Ordinogramme



Correction : Trace d'exécution avec les entrées 17 et 5

	a	b	q	r	r >= b
Début	17	5			
q = 0	17	5	0		
r = a	17	5	0	17	true
r = r-b	17	5	0	12	
q = q+1	17	5	1	12	true
r = r-b	17	5	1	7	
q = q+1	17	5	2	7	true
r = r-b	17	5	2	2	
q = q+1	17	5	3	2	false
Fin			3	2	

Sixième partie

Fonctions et procédures

Procédures

Fonctions
    Passage de paramètres

Factorisation de code

- Suites d'instructions utilisées en plusieurs endroits de l'algorithme
- Regrouper ce code en sous-programmes remplissant chacun une tâche simple
- Division logique de l'algorithme en composants
- Principe du « diviser pour mieux régner » : transformer un problème complexe en plusieurs sous-problèmes simples

Procédures

- Une procédure est un sous-programme simple réutilisable.
- Comme un algorithme, une procédure a un nom, un rôle et des déclarations.
- Définir une procédure :

```
function p ()
   -- role : faire des choses
   -- declarations : var : entier
   instructions
end
```

- Pour utiliser une procédure, on l'appelle par son nom suivi de parenthèses : `p()`.
- Une procédure devient une instruction.

Fonctions

Les procédures telles que nous venons de les définir ont un intérêt limité :

- pas d'entrées/sorties ;
- ne peut donc faire qu'une seule chose.

En ajoutant des entrées/sorties, on peut définir des fonctions.

Maths $f : \begin{array}{ccc} \mathbb{R} \times \mathbb{N} & \to & \mathbb{R} \\ (x, n) & \mapsto & x^n \end{array}$

- x et n sont les entrées (de types respectifs réel et naturel).
- la sortie est définie comme valant x^n
- on utilise la fonction par $f(\pi, 2)$

Fonctions : définition

Une fonction se comporte exactement comme un algorithme :

- elle comporte un entête contenant
  - son nom
  - son rôle
  - ses entrées (assorties de préconditions)
  - ses sorties (assorties de postconditions)
  - ses déclarations
- et un corps contenant les *statements* définissant le corps de la fonction.

Fonctions : syntaxe

- Définition d'une fonction nommée `f` prenant deux arguments

```
function f (arg1, arg2)
    -- role : ...
    -- entrees : arg1 : type1, arg2 : type2
    -- sorties : s : type
    -- declarations : ...
    Instructions
    return s
end
```

- Utilisation de cette fonction :
  - `var = f(1, 2)`
  - `var = f(x, y)`
  - `var = f(3*7-1, y)`
- Une fonction devient une expression.

Appelé, appelant

Lors de l'exécution d'un programme, l'appel à une fonction ou une procédure introduit un changement de contexte.

- Dans l'exemple précédent, la ligne `carre = puissance(x, 2)` appelle la fonction puissance.
- `puissance` est la *fonction appelée*.
- L'algorithme principal est nommé l'*appelant*.
- Une fonction peut en appeler une autre (ex : une fonction de calcul de n^n appellerait la fonction puissance).
- La fonction appelée est exécutée avec ses propres données et arguments.
- La fonction appelante ne connaît de la fonction exécutée que ce qu'elle retourne.

Portée des variables

Les fonctions et procédures ont leurs propres déclarations.

- Ces variables n'existent que pendant l'exécution de la méthode.
- Ces variables peuvent masquer une variable de l'algorithme ou de la fonction appelante.
- Contexte ou environnement d'exécution : l'ensemble des variables connues à l'intérieur de la fonction (les déclarations, entrées et sorties)
- Portée des variables : exemple.

Procédures et fonctions (le retour)

Une procédure est en fait une fonction sans sortie (c'est-à-dire sans `return`).

- Une procédure peut avoir des arguments.
- Une procédure est en réalité une instruction.
- Exemples : `print("Bonjour")`, `table.insert(t, 1)`, `string.gsub(ch, "a", "b")`

- Une fonction est en réalité une expression.
- Sa valeur et son type sont la valeur et le type de la variable de sortie.
- Le statement `return` doit renvoyer une expression du bon type.
- Exemple : `io.read()` est une fonction sans arguments qui renvoie une chaîne de caractères.
- Exemple : `string.sub(ch, 5, 6)` est une fonction de trois arguments qui renvoie une chaîne.

Exercice : arbre d'évaluation

- Quelle est la nature de `puissance(2+3, 7) + puissance(2*3, 5) + puissance(puissance(2, 3), 3)` ?
- Réaliser son arbre d'évaluation.

Paramètres et arguments

- Dans la définition d'une fonction, les arguments sont dénommés des paramètres formels.
- Lors de l'appel de la fonction, elle est appelée avec des paramètres effectifs : les valeurs sur lesquelles la fonction travaillera.
- Les paramètres formels peuvent « cacher » une variable de l'algorithme.
- Lors de l'exécution de la fonction, les paramètres formels sont remplacés par la valeur du paramètre effectif sur lequel la fonction est appelée.

Exemple `function f (x, y)`

- `x` et `y` sont les paramètres formels

- Que valent x et z après cette exécution ?

Passage par valeur, par copie, par référence

Transmission des paramètres :

- Par valeur (ou par copie) : le paramètre est transmis sous forme de copie, le modifier dans la fonction ne modifie pas le paramètre effectif.
- Par référence : le paramètre est transmis sous forme de pointeur (adresse mémoire). La fonction/procédure peut donc modifier le paramètre effectif (effets de bord).

Exemple `function f (x) x = 0 end`

Septième partie

Notes

Références, ressources

Lua5.1 Roberto Ierusalimschy, Luiz Henrique de Figueiredo et Waldemar Celes « Lua 5.1 Reference Manual » `http://www.lua.org/manual/5.1/`

PiLua Roberto Ierusalimschy « Programming in Lua » `http://www.lua.org/pil/`

AaP Christopher D. Pine et Jean-Pierre Anghel « Apprendre à programmer » `http://www.ruby-doc.org/docs/ApprendreProgrammer/Apprendre_a_Programmer.pdf` [6]

HTLCS Allen B. Downey « How to Think Like a Computer Scientist, Java Version » `http://openbookproject.net/thinkcs/java.php` [7]

LPTHW Zed A. Shaw « Learn Python the Hard Way » `http://learnpythonthehardway.org/` [8]

6. Pour le langage Ruby
7. Pour le langage Java
8. Pour le langage Python