# Programmer avec Orvet

(version 0)

Renaud Sirdey

*renaud.sirdey@gmail.com*

17 février 2016

*Dieu a créé les nombres entiers, le reste est l'œuvre de l'homme.*

L. Kronecker

# Table des matières

# 1. Introduction

Orvet est un langage de programmation, un vrai, destiné aux enfants disons à partir de la 6ème.

Il est intégralement en français !

Dans sa première version, Orvet permet de manipuler des nombres entiers (sans limitation de taille) ainsi que des booléens.

Pour les informaticiens pressés désireux de faire quelques rapides essais, aller à la section 17, page 55.

Pour les aspects installation et intégration avec Notepad++, aller à la section 14, page 39.

Pour commencer à rentrer tranquillement dans le langage, sans nécessairement programmer tout de suite, eh bien, il suffit de continuer la lecture de ce petit guide…

# 2. Un premier programme

En Orvet, tout est explicite.

Par exemple, le programme suivant déclare une variable entière nommée `nombre`, la lis (avec l'instruction `lire`) et affiche son contenu, c'est-à-dire la valeur qui vient d'être lue (avec l'instruction `montrer`).

```
$ Un premier programme en Orvet.

entier nombre

lire nombre

montrer nombre
```

Un programme Orvet peut contenir des commentaires (c'est même souhaitable) afin de clarifier son fonctionnement. Toute ligne commençant soit par le caractère `$`[1] (suivi d'un espace) soit par le caractère `#` (suivi d'un espace) est une ligne de commentaire.

Sans surprise, l'exécution de ce programme par l'interpréteur d'Orvet donne :

```
Orvet version 0.1
Chargement du programme
9 lignes chargées
Démarrage de l'exécution

Valeur de l'entier nombre ? 12
nombre = 12

Fin de l'exécution
```

[1] Car les commentaires valent de l'or ! C'est ce que pense Donald Knuth des équations ☺

Et voilà !

## 3. Déclaration et affectation de variables entières

Une variable entière est une sorte de boîte qui contient un nombre entier.

Une variable possède un nom (le nom de la boîte) et une valeur (le nombre contenu dans la boîte).

En Orvet, un nom de variable doit commencer par une lettre (possiblement accentuée, minuscule ou majuscule) qui peut être suivie d'une quelconque séquence mélangeant lettres (possiblement accentuées, minuscules ou majuscules), chiffres et caractère '_'.

Exemple de noms de variables valides (et invalides) :

```
a
abc
a123
un_nom_de_variable_très_long
A
Abc
ABC123
~~123~~
~~a+b~~
~~renaud.sirdey@gmail.com~~
```

Comme nous venons de le voir une variable entière doit être déclarée à l'aide de l'instruction `entier`. Une variable nouvellement déclarée vaut toujours 0.

Le programme suivant :

```
entier nombre

montrer nombre
```

donne donc :

```
Orvet version 0.1
Chargement du programme
5 lignes chargées
Démarrage de l'exécution

nombre = 0

Fin de l'exécution
```

L'instruction `montrer` permet, on l'a compris, d'afficher la valeur d'une variable – ici la variable `nombre`.

Pour affecter une nouvelle valeur à une variable on utilise l'instruction `affecter` comme illustré dans l'exemple de programme ci-après :

```
entier x
entier y

montrer x

affecter 123 à x

montrer x

montrer y

affecter x à y

montrer y
```

Programme dont l'exécution donne tout simplement :

```
Orvet version 0.1
Chargement du programme
15 lignes chargées
Démarrage de l'exécution

x = 0
x = 123
y = 0
y = 123

Fin de l'exécution
```

On peut donc affecter une valeur à une variable (instruction « affecter 123 à x » ci-dessus) ou affecter la valeur d'une variable à une autre variable (instruction « affecter x à y » ci-dessus).

Nous savons déclarer des variables entières, lire et écrire (instruction `montrer`) leurs valeurs ainsi qu'y affecter de nouvelles valeurs. Nous sommes maintenant prêts à calculer !

## 4. Faire des calculs

Pour faire des calculs, rien de plus simple ! Exemple :

```
entier x
entier y
entier z

lire x
lire y

ajouter x à y dans z
```

```
montrer z
```

Ce qui donne :

```
Orvet version 0.1
Chargement du programme
14 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 1234567890
Valeur de l'entier y ? 9876543210
z = 11111111100

Fin de l'exécution
```

Toutes les instructions de calcul ont le même format et les mêmes contraintes. On peut ajouter les valeurs de deux variables dans une variable (ce que nous venons de faire ci-dessus). Ajouter une valeur à la valeur d'une variable dans une variable, ajouter la valeur d'une variable à une valeur dans une variable ou encore ajouter deux valeurs dans une variable.

Exemple de toutes ces (autres) formes :

```
entier x

montrer x

ajouter 1 à x dans x

montrer x

ajouter x à 1 dans x

montrer x

ajouter 2 à 2 dans x

montrer x
```

Qui donne

```
Orvet version 0.1
Chargement du programme
22 lignes chargées
Démarrage de l'exécution

x = 0
x = 1
x = 2
x = 4
```

```
Fin de l'exécution
```

De manière analogue, en Orvet, on peut :

- **soustraire** <$var_1$|$val_1$> **à** <$var_2$|$val_2$> **dans** <var> ;
- **multiplier** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> ;
- **diviser** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> (<$var_2$|$val_2$> devant être différent de 0) ;
- **réduire** <$var_1$|$val_1$> **modulo** <$var_2$|$val_2$> **dans** <var> (<$var_2$|$val_2$> devant également être différent de 0) ;
- **élever** <$var_1$|$val_1$> **à la puissance** <$var_2$|$val_2$> **dans** <var> (<$var_2$|$val_2$> devant être positif ou nul).

La notation <var|val> signifie que l'opérande est soit une variable, soit une valeur. La notation <var> signifie que l'opérande ne peut être qu'une variable.

Pour illustrer cela, le programme ci-dessous réalise la division euclidienne de x par y :

```
entier x
entier y

lire x
lire y

entier quotient
entier reste

diviser x par y dans quotient

réduire x modulo y dans reste

montrer quotient
montrer reste
```

Qui donne par exemple :

```
Orvet version 0.1
Chargement du programme
18 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 17
Valeur de l'entier y ? 5
quotient = 3
reste = 2

Fin de l'exécution
```

C'est le moment de préciser que, pour un grand nombre de raisons (d'ailleurs assez profondes), l'exécution d'un programme peut parfois engendrer des erreurs. Par exemple, si l'on entre la valeur 0 pour l'entier y, on obtient :

```
Orvet version 0.1
Chargement du programme
18 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 17
Valeur de l'entier y ? 0
Erreur ligne 13 : division par zéro !
Erreur à la ligne 13 : instruction inconnue ou séquence interrompue

Fin de l'exécution
```

Attention à la soustraction : soustraire 2 **à** 3 donne 1 !

Autre exemple petit exemple élémentaire de calcul :

```
entier x
entier p

lire x

élever 2 à la puissance x dans p

montrer p
```

Qui donne

```
Orvet version 0.1
Chargement du programme
12 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 128
p = 340282366920938463463374607431768211456

Fin de l'exécution
```

Ce qui montre bien que $2^{128}$ est un grand nombre, et, de ce fait, qu'il n'y a pas de limitation de taille sur les entiers manipulés en Orvet.

Pour finir sur les instructions de calcul, deux instructions supplémentaires existent :

- **maximiser** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> ;
- **minimiser** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> ;

Ces deux instructions permettent d'affecter respectivement la plus grande ou la plus petite des valeurs des deux premiers opérandes dans la variable donnée par le troisième.

## 5. Une première structure de boucle

Nous allons maintenant construire des programmes un peu plus intéressants !

Pour ce faire, nous allons utiliser une première structure de boucle :

**pour** <var> **de** <$var_1$|$val_1$> **à** <$var_2$|$val_2$> **faire**

une séquence d'instructions

**fin**

Cette structure permet de répéter l'exécution de la séquence d'instructions que l'on appelle le *corps de la boucle* <$var_2$|$val_2$>- <$var_1$|$val_1$>+1 fois, <var> étant incrémentée de 1 à chaque tour de boucle.

De telles boucles peuvent être imbriquées, c'est-à-dire que le corps de la boucle peut lui aussi contenir des boucles.

Les quelques exemples suivants vont rendre tout ceci limpide.

Tout d'abord le programme le plus simple contenant une boucle :

```
entier N
entier i

lire N

pour i de 1 à N faire

    montrer i

fin
```

Programme dont l'exécution donne tout simplement et sans surprise :

```
Orvet version 0.1
Chargement du programme
13 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 10
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
```

```
i = 9
i = 10

Fin de l'exécution
```

Remarquons toutefois que si l'on entre un entier très grand pour N, un nombre à 25 chiffres fera largement l'affaire, le programme ci-dessus comptera pour ce qui ressemble fortement à l'éternité !

Illustrons tout de suite cette histoire de boucles imbriquées pour démystifier :

```
entier N
entier i
entier j

lire N

pour i de 1 à N faire

    montrer i

    pour j de i à N faire

        montrer j
    fin

fin
```

L'exécution donne alors :

```
Orvet version 0.1
Chargement du programme
19 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 5
i = 1
j = 1
j = 2
j = 3
j = 4
j = 5
i = 2
j = 2
j = 3
j = 4
j = 5
i = 3
j = 3
j = 4
j = 5
i = 4
```

```
j = 4
j = 5
i = 5
j = 5

Fin de l'exécution
```

On peut ainsi (en principe) imbriquer autant de niveaux de boucle que l'on souhaite. En pratique, néanmoins, s'il y a trop de niveaux de boucle imbriqués, le temps d'exécution du programme va généralement augmenter[2].

Bien entendu, les boucles ne servent pas qu'à afficher. Le programme suivant, par exemple, calcule la somme des entiers de 1 à N :

```
entier N
entier i
entier somme

lire N

pour i de 1 à N faire

    ajouter i à somme dans somme

fin

montrer somme
```

Ce qui donne :

```
Orvet version 0.1
Chargement du programme
16 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 100
somme = 5050

Fin de l'exécution
```

A ce stade, on peut se poser la question de savoir si l'on aurait pu faire un programme plus efficace, plus rapide ?

En fait, une formule bien connue pour calculer la somme des entiers de 1 à N est S=Nx(N+1)/2. L'avantage de cette formule est qu'elle se résume à 1 addition, 1 multiplication et 1 division

[2] Disons, en simplifiant, qu'il y a une relation entre le nombre de niveaux de boucle imbriqués et ce que l'on appelle la complexité de l'algorithme qui est une sorte de mesure abstraite du temps d'exécution. Je n'en dis pas plus. Pour l'instant nous apprenons à écrire des programmes et pas (encore) à les analyser.

par 2 alors que le programme ci-dessus nécessite de faire N additions, soit un nombre beaucoup plus grand d'opérations.

Réfléchissons quelques instants et vérifions la formule ci-dessus. Pour N=1 on a $S_1=1x(1+1)/2=1$, la formule fonctionne. Pour N=2 on a $S_2=2x(2+1)/2=3=1+2$, tout va bien. Supposons maintenant que la formule fonctionne également pour N-1, c'est-à-dire que $S_{N-1}=(N-1)x(N-1+1)/2=(N-1)N/2$. Alors qu'en est-il de $S_N$ ? $S_N=N+S_{N-1}=N+(N-1)N/2=(2N+(N-1)N)/2=(2N+N^2-N)/2=(N+N^2)/2=N(N+1)/2$. On appelle cela une preuve par récurrence[3].

En conséquence, le programme suivant est équivalent au programme précédent (qui avait surtout vocation à montrer comment faire des calculs dans une boucle…) :

```
entier N
entier somme

lire N

ajouter 1 à N dans somme

multiplier somme par N dans somme

diviser somme par 2 dans somme

montrer somme
```

Ce qui donne :

```
Orvet version 0.1
Chargement du programme
16 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 100
somme = 5050

Fin de l'exécution
```

Un affichage identique à celui du programme initial. En réfléchissant un peu, nous avons donc pu imaginer un programme beaucoup plus efficace qui se limite à ne faire que trois opérations pour calculer la somme des entiers de 1 à N et ce quel que soit N !

Moralité : il faut toujours réfléchir un peu avant d'écrire un programme…

Un dernier exemple simple qui va nous permettre de manipuler des grands nombres, le calcul du produit des nombres de 1 à N (on appelle cela le factoriel de N) :

```
entier N
```

[3] La notion de récurrence est très liée à la notion de récursivité abordée à la section 15 (pas la peine d'y aller trop vite néanmoins).

```
entier i
entier prod

lire N

affecter 1 à prod

pour i de 1 à N faire

    multiplier prod par i dans prod

fin

montrer prod
```

Qui donne,

```
Orvet version 0.1
Chargement du programme
18 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 128
prod = 38562048236258042173567706592346364061749310959022359027882840
327637340257516554356068616858850736153403005183305891634759217293 2
624988577661149552450393577600346447092792476924955852800000000000000
0000000000000000000

Fin de l'exécution
```

Un exemple de nombre astronomiquement grand s'il en est ! Beaucoup plus grand que $2^{128}$ que nous avons calculé précédemment et beaucoup plus grand qu'un Gogol qui vaut « juste » $10^{100}$.

## 6. Tirer au hasard

Dans certains programmes, il est parfois fort utile de pouvoir choisir des nombres au hasard[4].

En Orvet on tire au hasard à l'aide de l'instruction :

**tirer** <var> **au hasard entre** <$var_1$|$val_1$> **et** <$var_2$|$val_2$>

Ainsi <var> se retrouvera affectée avec une quelconque des <$var_2$|$val_2$>- <$var_1$|$val_1$>+1 valeurs entre <$var_2$|$val_2$> et <$var_1$|$val_1$> (inclusivement, donc). Chacune de ces valeurs possède des chances identiques de sortir (on dit qu'elles sont équiprobables).

[4] Il y a des raisons très profondes qui font que les ordinateurs ne peuvent intrinsèquement pas produire du hasard et il y a également d'autres raisons tout aussi profondes qui montrent qu'ils peuvent produire un pseudo-hasard très convaincant. Ces liens entre ordinateurs et hasard sont parmi les sujets d'étude les plus intéressant en informatique, mais là je digresse quelque peu…

Encore une fois, démystifions à l'aide de l'exemple simple d'un programme qui réalise l'équivalent de lancers d'un dé :

```
entier N
entier i
entier x

lire N

pour i de 1 à N faire

    tirer x au hasard entre 1 et 6

    montrer x

fin
```

Voici quelques lancers :

```
Orvet version 0.1
Chargement du programme
18 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 10
x = 3
x = 6
x = 6
x = 2
x = 5
x = 5
x = 6
x = 3
x = 6
x = 1

Fin de l'exécution
```

Et si l'on relance le programme, il y a donc de très fortes chances pour que l'on obtienne une séquence différente de valeurs[5] :

```
Orvet version 0.1
Chargement du programme
18 lignes chargées
Démarrage de l'exécution

Valeur de l'entier N ? 10
x = 3
x = 2
```

[5] Une fois tirée une première séquence de N valeurs, on a une chance sur $6^N$ de tirer la même séquence aux N lancers suivants. Pour N=10, cela donne 1 chance sur 60466176.

```
x = 5
x = 1
x = 1
x = 3
x = 2
x = 2
x = 2
x = 3

Fin de l'exécution
```

## 7. Amélioration de l'affichage

L'instruction `écrire` va nous permettre de réaliser un affichage plus riche qu'avec l'instruction `montrer` que nous avons utilisée jusqu'à présent.

Lorsque l'on apprend un nouveau langage de programmation, le premier programme que l'on écrit est un programme qui affiche « Bonjour monde ! ». Ainsi l'exige la tradition !

Voici ce programme en Orvet :

```
écrire Bonjour monde !
```

Sans surprise particulière, voici le résultat de son exécution :

```
Orvet version 0.1
Chargement du programme
4 lignes chargées
Démarrage de l'exécution

Bonjour monde !

Fin de l'exécution
```

Plus généralement cette instruction imprime tous ce qui se trouve après le mot-clef `écrire` et remplace les noms de variables précédés du caractère $ par les valeurs de ces variables.

Exemple où l'on reprend notre programme pour la division euclidienne (sect. 4) en le dotant d'un affichage plus intelligible :

```
entier x
entier y

lire x
lire y

entier quotient
entier reste
```

```
diviser x par y dans quotient

réduire x modulo y dans reste

écrire Le quotient de la division de $x par $y est $quotient
écrire Le reste de la division de $x par $y est $reste
```

D'où :

```
Orvet version 0.1
Chargement du programme
19 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 17
Valeur de l'entier y ? 5
Le quotient de la division de 17 par 5 est 3
Le reste de la division de 17 par 5 est 2

Fin de l'exécution
```

Tout ceci étant vu, nous allons maintenant nous attaquer à un autre pilier de la programmation, les *booléens*, qui vont nous offrir des possibilités quasiment sans limites[6].

## 8. Déclarer et affecter des variables booléennes

Une variable booléenne c'est la même chose qu'une variable entière, c'est une sorte de boîte qui contient une valeur, à ceci près qu'elle ne contient pas une valeur numérique (un nombre) mais ce que l'on appelle une *valeur de vérité* (`vrai` ou `faux`).

Les variables booléennes vont nous permettre de tester des conditions qui vont influer sur l'exécution de nos programmes.

Avant de voir cela, commençons par les bases. Pour déclarer une variable booléenne, on utilise l'instruction `booléen`, les variables booléennes étant par défaut initialisées à `faux`. Enfin, pour y affecter une valeur (qui sera soit `vrai`, soit `faux`, soit la valeur de vérité d'une autre variable booléenne) on utilise l'instruction `affecter` de manière analogue au cas des variables entières.

Exemple :

```
booléen a

montrer a

affecter vrai à a
```

[6] Autres que celles de la calculabilité. Non je ne digresse pas plus ici…

```
montrer a

booléen b

montrer b

affecter a à b

montrer b

affecter faux à a

montrer a
montrer b
```

Petit programme dont l'exécution donne :

```
Orvet version 0.1
Chargement du programme
24 lignes chargées
Démarrage de l'exécution

a = faux
a = vrai
b = faux
b = vrai
a = faux
b = vrai

Fin de l'exécution
```

## 9. Comparer des entiers

Maintenant que nous savons déclarer et affecter des variables booléennes, nous allons pouvoir les utiliser pour stocker les résultats de comparaisons entre des entiers.

Les instructions `comparer` et `différencier` sont les premières instructions de ce type.

L'instruction

**comparer** <$var_1$|$val_1$> **avec** <$var_2$|$val_2$> **dans** <var>

mettra `vrai` dans <var> si la valeur entière <$var_1$|$val_1$> est égale à la valeur entière <$var_2$|$val_2$>, et `faux` sinon.

Quant à elle, l'instruction

**différencier** <$var_1$|$val_1$> **avec** <$var_2$|$val_2$> **dans** <var>

fera le contraire, elle mettra `vrai` dans <var> si la valeur entière <$var_1$|$val_1$> est différente de la valeur entière <$var_2$|$val_2$>, et `faux` sinon.

Pas si compliqué que ça ! Exemple :

```
entier x
entier y

lire x
lire y

booléen égaux
booléen différents

comparer x avec y dans égaux

différencier x avec y dans différents

montrer égaux
montrer différents
```

Ce qui donne :

```
Orvet version 0.1
Chargement du programme
19 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
Valeur de l'entier y ? 11
égaux = faux
différents = vrai

Fin de l'exécution
```

Ou encore :

```
Orvet version 0.1
Chargement du programme
19 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
Valeur de l'entier y ? 10
égaux = vrai
différents = faux

Fin de l'exécution
```

En plus, les deux instructions supplémentaires suivantes :

- **minorer** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var>.
- **majorer** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var>.

permettent de tester si <$var_2$|$val_2$> est inférieur (strictement) à <$var_1$|$val_1$>, pour la première, et si <$var_2$|$val_2$> est supérieur (strictement) à <$var_1$|$val_1$>, pour la seconde.

## 10. Et si ?

Nous allons maintenant voir comment ces booléens peuvent influer sur l'exécution de nos programmes.

Pour ce faire, nous sommes maintenant en mesure d'introduire une structure d'exécution conditionnelle :

**si** <var> **alors**

    une séquence d'instructions

**fin**

Structure qui permet d'exécuter la séquence d'instructions si la valeur de vérité de la variable booléenne <var> est `vrai` et de ne pas l'exécuter sinon.

Exemple à partir du programme précédent que l'on va doter d'une sortie un peu plus intelligible :

```
entier x
entier y

lire x
lire y

booléen égaux
booléen différents

comparer x avec y dans égaux

différencier x avec y dans différents

si égaux alors

    écrire Les deux nombres sont égaux

fin

si différents alors

    écrire Les deux nombres sont différents

fin
```

D'où :

```
Orvet version 0.1
Chargement du programme
```

```
29 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
Valeur de l'entier y ? 11
Les deux nombres sont différents

Fin de l'exécution
```

Et :

```
Orvet version 0.1
Chargement du programme
29 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
Valeur de l'entier y ? 10
Les deux nombres sont égaux

Fin de l'exécution
```

L'exemple suivant, qui n'est guère plus compliqué, permet de répondre à la question de savoir si un entier x est un multiple d'un autre entier y.

```
entier x
entier y

lire x
lire y

entier r

booléen b

réduire x modulo y dans r

comparer r avec 0 dans b

si b alors

    écrire $x est bien un multiple de $y

fin

différencier r avec 0 dans b

si b alors

    écrire $x n'est pas un multiple de $y
    écrire Le reste de la division de $x par $y est $r

fin
```

Programme dont l'exécution donne par exemple :

```
Orvet version 0.1
Chargement du programme
31 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 16
Valeur de l'entier y ? 4
16 est bien un multiple de 4

Fin de l'exécution
```

Ou encore, pour illustrer le second cas :

```
Orvet version 0.1
Chargement du programme
31 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 17
Valeur de l'entier y ? 4
17 n'est pas un multiple de 4
Le reste de la division de 17 par 4 est 1

Fin de l'exécution
```

Enfin, ces structures d'exécution conditionnelle se combinent tout à fait entres elles ainsi qu'avec les structures de boucle. Voici un exemple un peu plus complexe qui donne tous les diviseurs d'un nombre et qui indique à la fin si ce nombre est premier (rappelons qu'un nombre est premier s'il n'a de diviseurs autres que 1 et lui-même). Nous avons ajouté les numéros de lignes pour faciliter l'explication du fonctionnement de ce programme donnée un peu plus loin.

```
04. entier x
05.
06. lire x
07.
08. entier y
09. entier i
10. entier reste
11. booléen juste
12. booléen premier
13.
14. soustraire 1 à x dans y
15.
16. affecter vrai à premier
17.
18. pour i de 2 à y faire
```

```
19.
20.     réduire x modulo i dans reste
21.
22.     comparer reste avec 0 dans juste
23.
24.     si juste alors
25.
26.         écrire $i est un diviseur de $x
27.
28.         affecter faux à premier
29.
30.     fin
31.
32. fin
33.
34. si premier alors
35.
36.     écrire $x est un nombre premier
37.
38. fin
```

Testons ce programme !

```
Orvet version 0.1
Chargement du programme
39 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 20
2 est un diviseur de 20
4 est un diviseur de 20
5 est un diviseur de 20
10 est un diviseur de 20

Fin de l'exécution
```

Et, cette fois avec un nombre premier en entrée :

```
Orvet version 0.1
Chargement du programme
39 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 23
23 est un nombre premier

Fin de l'exécution
```

Nos petits programmes commencent à devenir intéressants !

Détaillons donc un peu le fonctionnement du programme ci-dessus.

Le programme demande initialement la valeur du nombre à tester à l'utilisateur (ligne 6) comme nous l'avons déjà fait de multiples fois. Ce nombre est stocké dans la variable `x`. Ligne 14, on met `x`-1 dans la variable `y`. Par exemple, si `x` vaut 20, alors `y` vaudra 19. Cela permet, ligne 18, de faire une boucle sur toutes les valeurs de 2 à `x`-1. Suivant le même exemple, si `x` vaut initialement 20 alors `i` vaudra donc successivement 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 et 19. Avant l'exécution de la boucle, la variable booléenne `premier` est mise à `vrai` (ligne 16), c'est-à-dire que l'on fait au départ l'hypothèse que le nombre testé est premier et c'est le travail de la boucle de trouver justement des diviseurs qui, s'il en existe au moins un, indiqueront que le nombre testé x n'est pas premier.

Alors allons-y, bouclons !

La première instruction du corps de boucle, ligne 20, calcule le reste de la division de `x` par `i`. Pour notre exemple, avec `x` à 20, le premier reste est 0 puisque 20 est bien entendu divisible par 2. L'instruction de comparaison de la ligne 22 met `vrai` dans la variable booléenne `juste` si le reste en question est 0 et `faux` sinon. Pour `x` à 20 et `i` à 2, `juste` prend donc `vrai`. Ainsi, le test de la ligne 24 est satisfait et les instructions des lignes 26 et 28 sont exécutées. Le programme indique alors que 2 est un diviseur de 20 et la variable booléenne `premier` prend `faux`.

À l'itération suivante de la boucle, `x` à 20 et `i` à 3, les choses se répètent, exception faite que le reste calculé à la ligne 20 n'est cette fois pas 0 (puisque 20 n'est pas divisible par 3). Les instructions des lignes 26 et 28 ne sont cette fois donc pas exécutées, ce qui signifie que la variable booléenne `premier` reste inchangée (elle valait `faux` puisqu'un diviseur – 2 – a été trouvé à l'itération précédente, elle reste donc à `faux`, tout va bien).

L'exécution se répète ainsi pour toutes les valeurs 2 à 19, avec une exécution des lignes 26 et 28 pour les valeurs 2 (nous l'avons déjà vu), 4, 5 et 10 de `i`. Ceci a ainsi pour effet de mettre 4 fois la valeur `faux` dans `premier`, ce qui est amplement suffisant pour cette variable soit à `faux` à la fin de l'exécution de la boucle résultant en une absence d'exécution de l'instruction de la ligne 36. In fine, le programme, à juste titre, n'indiquera donc pas que 20 est premier.

A contrario, pour un nombre premier en entrée, par exemple 23, alors aucune des valeurs de `i` successives 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22 n'engendrera de reste nul à la ligne 20. Ainsi, les instructions des lignes 26 et 28 ne seront pas exécutées et, en particulier, la variable booléenne `premier` ne prendra jamais `faux`. À la fin de la boucle, l'instruction de la ligne 36 sera exécutée et le programme indiquera par ce biais, et à juste titre, que 23 est un nombre premier.

Lors de l'exécution d'un programme Orvet, il est possible de demander à ce que chaque instruction imprime une trace de son exécution (section 13, page 37). Cela permet de mieux comprendre le fonctionnement d'un programme lorsque c'est nécessaire.

Pour le programme ci-dessus, cette exécution avec trace donne (pour 20 en entrée) :

```
Orvet version 0.1
Chargement du programme
```

```
39 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier x (initialisé à 0)
Valeur de l'entier x ? 20
6 - Lecture de la valeur de l'entier x
8 - Définition de l'entier y (initialisé à 0)
9 - Définition de l'entier i (initialisé à 0)
10 - Définition de l'entier reste (initialisé à 0)
11 - Définition du booléen juste (initialisé à faux)
12 - Définition du booléen premier (initialisé à faux)
14 - Soustraction de la valeur 1 à la valeur 20 dans la variable entière y
16 - Affectation de la valeur vrai à la variable booléenne premier
18 - Bouclage sur i de 2 à 19
20 - Réduction de la valeur 20 modulo la valeur 2 dans la variable entière reste
22 - Comparaison des valeurs 0 et 0 dans la variable booléenne juste
24 - Condition sur juste à vrai
2 est un diviseur de 20
26 - Ecriture ci-dessus
28 - Affectation de la valeur faux à la variable booléenne premier
20 - Réduction de la valeur 20 modulo la valeur 3 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 4 dans la variable entière reste
22 - Comparaison des valeurs 0 et 0 dans la variable booléenne juste
24 - Condition sur juste à vrai
4 est un diviseur de 20
26 - Ecriture ci-dessus
28 - Affectation de la valeur faux à la variable booléenne premier
20 - Réduction de la valeur 20 modulo la valeur 5 dans la variable entière reste
22 - Comparaison des valeurs 0 et 0 dans la variable booléenne juste
24 - Condition sur juste à vrai
5 est un diviseur de 20
26 - Ecriture ci-dessus
28 - Affectation de la valeur faux à la variable booléenne premier
20 - Réduction de la valeur 20 modulo la valeur 6 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 7 dans la variable entière reste
22 - Comparaison des valeurs 6 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 8 dans la variable entière reste
22 - Comparaison des valeurs 4 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 9 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 10 dans la variable entière reste
22 - Comparaison des valeurs 0 et 0 dans la variable booléenne juste
24 - Condition sur juste à vrai
10 est un diviseur de 20
26 - Ecriture ci-dessus
28 - Affectation de la valeur faux à la variable booléenne premier
20 - Réduction de la valeur 20 modulo la valeur 11 dans la variable entière reste
22 - Comparaison des valeurs 9 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 12 dans la variable entière reste
22 - Comparaison des valeurs 8 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 13 dans la variable entière reste
22 - Comparaison des valeurs 7 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 14 dans la variable entière reste
22 - Comparaison des valeurs 6 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 15 dans la variable entière reste
22 - Comparaison des valeurs 5 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
```

```
20 - Réduction de la valeur 20 modulo la valeur 16 dans la variable entière reste
22 - Comparaison des valeurs 4 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 17 dans la variable entière reste
22 - Comparaison des valeurs 3 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 18 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 20 modulo la valeur 19 dans la variable entière reste
22 - Comparaison des valeurs 1 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
34 - Condition sur premier à faux
```

Où, on l'aura compris, les traces des exécutions successives du corps de boucle sont alternativement en vert et en orange.

Et, cette fois avec 23 en entrée :

```
Orvet version 0.1
Chargement du programme
39 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier x (initialisé à 0)
Valeur de l'entier x ? 23
6 - Lecture de la valeur de l'entier x
8 - Définition de l'entier y (initialisé à 0)
9 - Définition de l'entier i (initialisé à 0)
10 - Définition de l'entier reste (initialisé à 0)
11 - Définition du booléen juste (initialisé à faux)
12 - Définition du booléen premier (initialisé à faux)
14 - Soustraction de la valeur 1 à la valeur 23 dans la variable entière y
16 - Affectation de la valeur vrai à la variable booléenne premier
18 - Bouclage sur i de 2 à 22
20 - Réduction de la valeur 23 modulo la valeur 2 dans la variable entière reste
22 - Comparaison des valeurs 1 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 3 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 4 dans la variable entière reste
22 - Comparaison des valeurs 3 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 5 dans la variable entière reste
22 - Comparaison des valeurs 3 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 6 dans la variable entière reste
22 - Comparaison des valeurs 5 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 7 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 8 dans la variable entière reste
22 - Comparaison des valeurs 7 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 9 dans la variable entière reste
22 - Comparaison des valeurs 5 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 10 dans la variable entière reste
22 - Comparaison des valeurs 3 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 11 dans la variable entière reste
22 - Comparaison des valeurs 1 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
```

```
20 - Réduction de la valeur 23 modulo la valeur 12 dans la variable entière reste
22 - Comparaison des valeurs 11 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 13 dans la variable entière reste
22 - Comparaison des valeurs 10 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 14 dans la variable entière reste
22 - Comparaison des valeurs 9 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 15 dans la variable entière reste
22 - Comparaison des valeurs 8 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 16 dans la variable entière reste
22 - Comparaison des valeurs 7 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 17 dans la variable entière reste
22 - Comparaison des valeurs 6 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 18 dans la variable entière reste
22 - Comparaison des valeurs 5 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 19 dans la variable entière reste
22 - Comparaison des valeurs 4 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 20 dans la variable entière reste
22 - Comparaison des valeurs 3 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 21 dans la variable entière reste
22 - Comparaison des valeurs 2 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
20 - Réduction de la valeur 23 modulo la valeur 22 dans la variable entière reste
22 - Comparaison des valeurs 1 et 0 dans la variable booléenne juste
24 - Condition sur juste à faux
34 - Condition sur premier à vrai
23 est un nombre premier
36 - Ecriture ci-dessus
```

Nous allons maintenant introduire la dernière structure de boucle supportée par Orvet.

## 11. Boucle « tant que »

Jusqu'à présent, avec la structure de boucle que nous avons vue, nous ne pouvons boucler qu'un nombre prédéterminé de fois.

Avec la structure de boucle que nous allons maintenant introduire, il devient possible de boucler tant qu'une condition reste satisfaite.

En Orvet, on exprime une telle structure de la manière suivante :

**tant que** <var> **faire**

une séquence d'instructions

**fin**

Ceci permet de répéter l'exécution de la séquence d'instructions que l'on appelle (toujours) le *corps de la boucle* tant que la variable booléenne <var> est à `vrai`. Ceci suppose que cette variable soit à `vrai` avant l'exécution de la boucle (sinon le corps de boucle n'est jamais exécuté – ce qui est parfois ce que l'on veut faire) puis qu'ultérieurement la séquence

d'instructions du corps de boucle finisse par passer <var> à `faux` (sinon quoi la boucle ne terminera jamais – ce qui est aussi parfois ce que l'on veut faire[7]).

Par exemple, pour se fixer les idées, nous souhaitons modifier le programme de la section précédente afin qu'il s'arrête dès qu'il trouve un diviseur du nombre d'entrée, au lieu d'énumérer tous ses diviseurs. Trouver un diviseur est un contre-exemple pour la primalité et permet au programme de conclure que le nombre d'entrée n'est pas premier.

Voici donc un programme qui réalise cela :

```
04. entier x
05.
06. lire x
07.
08. entier i
09. entier reste
10. booléen juste
11. booléen continuer
12. booléen premier
13. booléen divisible
14.
15. affecter 1 à i
16. affecter vrai à continuer
17.
18. tant que continuer faire
19.
20.    ajouter 1 à i dans i
21.
22.    réduire x modulo i dans reste
23.
24.    différencier reste avec 0 dans continuer
25.
26. fin
27.
28. comparer i avec x dans premier
29.
30. différencier i avec x dans divisible
31.
32. si premier alors
33.
34.    écrire $x est un nombre premier
35.
36. fin
37.
38. si divisible alors
39.
40.    écrire $x n'est pas premier, il est divisible par $i
41.
42. fin
```

[7] Néanmoins assez rarement en Orvet !

Essayons donc ce programme :

```
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 20
20 n'est pas premier, il est divisible par 2

Fin de l'exécution
```

Et avec un nombre justement premier :

```
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 23
23 est un nombre premier

Fin de l'exécution
```

Pour bien comprendre, détaillons donc un peu le fonctionnement de ce dernier programme.

Juste avant la boucle, ligne 15, on affecte 1 à `i`. Ligne 16, on affecte la valeur de vérité `vrai` à la variable booléenne `continuer` qui sert de condition pour la boucle de la ligne 18.

Ainsi, puisque que `continuer` est initialement `vrai`, le corps de boucle peut s'exécuter une première fois.

Le travail de la boucle est de tester des diviseurs possibles pour `x` et de s'arrêter dès qu'un diviseur est trouvé. Par exemple si `x` vaut 20, la première instruction de la boucle (ligne 20) ajoute 1 à `i` (qui vaut initialement 1) – et met donc 2 dans `i` – et l'instruction de la ligne 22 calcule donc le reste de la division de 20 par 2, la variable `reste` prend alors 0 puisque 20 est (évidemment !) divisible par 2. Ainsi, `continuer` prend `faux` à la ligne 24 et la boucle s'arrête avec la valeur 2 dans `i`.

In fine, après la boucle, 2 (dans `i`) est différent de 20 (dans `x`). Donc `premier` prend `faux` (ligne 28) et `divisible` prend `vrai` (ligne 30). C'est donc la condition de la ligne 38 qui est satisfaite et l'instruction d'écriture de la ligne 40 qui est exécutée. Le programme déclare donc à juste titre que 20 n'est pas premier.

Dans le cas d'un nombre premier, par exemple si `x` avait initialement valu 23, alors la première valeur de `i` à produire un reste nul à la ligne 22 aurait (par définition d'un nombre premier)

été justement 23[8]. Ainsi, la boucle se serait cette fois arrêtée avec la valeur 23 dans `i`. In fine, après la boucle, 23 (dans `i`) est égal à 23 (dans `x`). Donc `premier` prend `vrai` (ligne 28) et `divisible` prend `faux` (ligne 30). C'est donc la condition de la ligne 32 qui est satisfaite et l'instruction d'écriture de la ligne 34 qui est exécutée. Le programme déclare donc à juste titre que 23 est un nombre premier !

Les deux exécutions avec instructions tracées ci-dessous sous fournies pour illustrer le fonctionnement du programme en détails.

D'abord avec 20 :

```
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier x (initialisé à 0)
Valeur de l'entier x ? 20
6 - Lecture de la valeur de l'entier x
8 - Définition de l'entier i (initialisé à 0)
9 - Définition de l'entier reste (initialisé à 0)
10 - Définition du booléen juste (initialisé à faux)
11 - Définition du booléen continuer (initialisé à faux)
12 - Définition du booléen premier (initialisé à faux)
13 - Définition du booléen divisible (initialisé à faux)
15 - Affectation de la valeur 1 à la variable entière i
16 - Affectation de la valeur vrai à la variable booléenne continuer
18 - Bouclage sur continuer
20 - Addition des valeurs 1 et 1 dans la variable entière i
22 - Réduction de la valeur 20 modulo la valeur 2 dans la variable entière reste
24 - Test de la non égalité des valeurs 0 et 0 dans la variable booléenne continuer
28 - Comparaison des valeurs 2 et 20 dans la variable booléenne premier
30 - Test de la non égalité des valeurs 2 et 20 dans la variable booléenne
divisible
32 - Condition sur premier à faux
38 - Condition sur divisible à vrai
20 n'est pas premier, il est divisible par 2
40 - Ecriture ci-dessus

Fin de l'exécution
```

Comme prévu, le corps boucle (en vert) n'est donc exécuté qu'une seule fois.

Avec 23 en entrée, les choses se passent très différemment :

```
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier x (initialisé à 0)
Valeur de l'entier x ? 23
6 - Lecture de la valeur de l'entier x
8 - Définition de l'entier i (initialisé à 0)
9 - Définition de l'entier reste (initialisé à 0)
10 - Définition du booléen juste (initialisé à faux)
11 - Définition du booléen continuer (initialisé à faux)
12 - Définition du booléen premier (initialisé à faux)
```

[8] Certes, comme chacun sait, lorsque l'on teste la primalité de cette façon, on peut s'arrêter de chercher des diviseurs dès que l'on dépasse la racine carrée du nombre testé.

```
13 - Définition du booléen divisible (initialisé à faux)
15 - Affectation de la valeur 1 à la variable entière i
16 - Affectation de la valeur vrai à la variable booléenne continuer
18 - Bouclage sur continuer
20 - Addition des valeurs 1 et 1 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 2 dans la variable entière reste
24 - Test de la non égalité des valeurs 1 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 2 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 3 dans la variable entière reste
24 - Test de la non égalité des valeurs 2 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 3 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 4 dans la variable entière reste
24 - Test de la non égalité des valeurs 3 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 4 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 5 dans la variable entière reste
24 - Test de la non égalité des valeurs 3 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 5 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 6 dans la variable entière reste
24 - Test de la non égalité des valeurs 5 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 6 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 7 dans la variable entière reste
24 - Test de la non égalité des valeurs 2 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 7 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 8 dans la variable entière reste
24 - Test de la non égalité des valeurs 7 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 8 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 9 dans la variable entière reste
24 - Test de la non égalité des valeurs 5 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 9 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 10 dans la variable entière reste
24 - Test de la non égalité des valeurs 3 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 10 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 11 dans la variable entière reste
24 - Test de la non égalité des valeurs 1 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 11 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 12 dans la variable entière reste
24 - Test de la non égalité des valeurs 11 et 0 dans la variable booléenne
continuer
20 - Addition des valeurs 1 et 12 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 13 dans la variable entière reste
24 - Test de la non égalité des valeurs 10 et 0 dans la variable booléenne
continuer
20 - Addition des valeurs 1 et 13 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 14 dans la variable entière reste
24 - Test de la non égalité des valeurs 9 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 14 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 15 dans la variable entière reste
24 - Test de la non égalité des valeurs 8 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 15 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 16 dans la variable entière reste
24 - Test de la non égalité des valeurs 7 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 16 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 17 dans la variable entière reste
24 - Test de la non égalité des valeurs 6 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 17 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 18 dans la variable entière reste
24 - Test de la non égalité des valeurs 5 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 18 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 19 dans la variable entière reste
24 - Test de la non égalité des valeurs 4 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 19 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 20 dans la variable entière reste
24 - Test de la non égalité des valeurs 3 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 20 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 21 dans la variable entière reste
24 - Test de la non égalité des valeurs 2 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 21 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 22 dans la variable entière reste
```

```
24 - Test de la non égalité des valeurs 1 et 0 dans la variable booléenne continuer
20 - Addition des valeurs 1 et 22 dans la variable entière i
22 - Réduction de la valeur 23 modulo la valeur 23 dans la variable entière reste
24 - Test de la non égalité des valeurs 0 et 0 dans la variable booléenne continuer
28 - Comparaison des valeurs 23 et 23 dans la variable booléenne premier
30 - Test de la non égalité des valeurs 23 et 23 dans la variable booléenne
divisible
32 - Condition sur premier à vrai
23 est un nombre premier
34 - Ecriture ci-dessus
38 - Condition sur divisible à faux

Fin de l'exécution
```

Où les exécutions successives du corps de boucle ont de nouveau été montrées alternativement en vert et en orange.

## 12. Opérateurs logiques

Pour les plus avancés, Orvet supporte les opérations suivantes :

- **`complémenter`** <var$_1$|val$_1$> **`dans`** <var> (booléen → booléen) – NON logique.
- **`disjoindre`** <var$_1$|val$_1$> **`et`** <var$_2$|val$_2$> **`dans`** <var> (booléen × booléen → booléen) – OU logique.
- **`conjoindre`** <var$_1$|val$_1$> **`et`** <var$_2$|val$_2$> **`dans`** <var> (booléen × booléen → booléen) – ET logique.

La première de ces instructions – `complémenter` – réalise ce que l'on appelle un NON-logique : si l'entrée <var$_1$|val$_1$> vaut `vrai` alors <var> vaudra `faux` et si <var$_1$|val$_1$> vaut `faux` alors <var> vaudra `vrai`.

Le tableau suivant – on appelle cela une table de vérité – résume son fonctionnement :

| <var$_1$\|val$_1$> | <var> |
|---|---|
| faux | vrai |
| vrai | faux |

L'instruction `disjoindre`, quant à elle, implémente un OU-logique : <var> ne vaudra `faux` que si <var$_1$|val$_1$> et <var$_2$|val$_2$> valent tous les deux `faux`, sinon <var> vaudra `vrai`. Sa table de vérité donne donc :

| <var$_1$\|val$_1$> | <var$_2$\|val$_2$> | <var> |
|---|---|---|
| Faux | faux | faux |
| Faux | vrai | vrai |
| Vrai | faux | vrai |
| vrai | vrai | vrai |

Enfin, l'instruction conjoindre implémente ce que l'on appelle un ET-logique : <var> ne vaudra `vrai` que si <var$_1$|val$_1$> et <var$_2$|val$_2$> valent tous les deux `vrai`, sinon <var> vaudra `faux`. Sa table de vérité donne donc :

| <var$_1$\|val$_1$> | <var$_2$\|val$_2$> | <var> |
|---|---|---|
| faux | faux | faux |
| faux | vrai | faux |
| vrai | faux | faux |
| vrai | vrai | vrai |

Le petit exemple suivant illustre comment utiliser ces instructions :

```
booléen a
booléen b
booléen c

montrer a
montrer b

disjoindre a et b dans c

montrer c

complémenter a dans a

montrer a
montrer b

disjoindre a et b dans c

montrer c

conjoindre a et b dans c

montrer c

complémenter b dans b

montrer a
montrer b

conjoindre a et b dans c

montrer c
```

L'exécution de ce programme donne :

```
Orvet version 0.1
Chargement du programme
35 lignes chargées
Démarrage de l'exécution

a = faux
b = faux
```

```
c = faux
a = vrai
b = faux
c = vrai
c = faux
a = vrai
b = vrai
c = vrai

Fin de l'exécution
```

Ces opérateurs logiques un peu plus avancés permettent de faire obéir nos programmes – via les structures « si … alors » (section 10, page 23) et « tant que … faire » (section 11, page 30) – à des conditions aussi complexes qu'on le souhaite !

## 13. Arrêt et vérification de conditions

Deux instructions supplémentaires existent également :

- **arrêter**.
- **vérifier** <var> (booléen).

L'instruction `arrêter` permet simplement d'arrêter l'exécution du programme.

Par exemple, le programme qui teste la primalité d'un entier donné (section 11) peut se réécrire comme suit à l'aide de cette instruction :

```
entier x

lire x

entier y
entier i
entier reste
booléen juste

soustraire 1 à x dans y

pour i de 2 à y faire

    réduire x modulo i dans reste

    comparer reste avec 0 dans juste

    si juste alors

        écrire $x n'est pas premier, il est divisible par $i

        arrêter

    fin
```

```
fin

écrire $x est un nombre premier
```

Donc sous une forme un peu plus simple. Nous faisons grâce de l'exécution.

Enfin l'instruction `vérifier` permet d'arrêter le programme si une variable booléenne est à `faux`.

Exemple :

```
entier x
entier y
entier z

booléen vérif_div_0

lire x
lire y

différencier y avec 0 dans vérif_div_0

vérifier vérif_div_0

diviser x par y dans z

montrer z
```

Dont l'exécution donne :

```
Orvet version 0.1
Chargement du programme
21 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
Valeur de l'entier y ? 5
z = 2

Fin de l'exécution
```

Ou encore (cas le plus intéressant…) :

```
Orvet version 0.1
Chargement du programme
21 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 10
```

```
Valeur de l'entier y ? 0
Condition ligne 15 non vérifiée !

Fin de l'exécution
```

De quoi s'assurer du bon déroulement de l'exécution d'un programme Orvet !

## 14. Procédures et appels de procédures

Pour faire une bonne galette des rois il faut deux pâtes feuilletées et une bonne dose de crème d'amande. Une fois en possession de ces trois éléments, on commence par étaler la crème d'amande sur la première pâte feuilletée (attention il faut bien étaler quasiment jusqu'au bord pour éviter le syndrome de la fin de part trop sèche[9]). On place ensuite la fève, puis on recouvre avec la seconde pâte feuilletée. Pour finaliser, faire des entailles au couteau sur le dessus de la galette (on peut même s'essayer au dessin) et badigeonner de jaune d'œuf (c'est pour avoir l'effet doré à la cuisson). Enfin, faire des entailles régulièrement espacées au couteau sur le pourtour de la galette et le badigeonner de blanc d'œuf (qui jouera le rôle de colle à la cuisson). On enfourne pour 30 minutes à 180° dans un four préchauffé et le tour est joué. Facile !

Quel est le rapport entre la recette ci-dessus et la programmation en Orvet ?

Eh bien on peut commencer par remarquer que la recette ci-dessus est incomplète : je ne vous ai pas expliqué ni comment faire la pâte feuilletée, ni comment faire la crème d'amande. Difficile, donc, de faire une galette à ce stade.

Il se trouve que la pâte feuilletée intervient dans un grand nombre de recettes donc afin d'éviter de répéter sa description un grand nombre de fois (ce qui aurait pour effet d'augmenter significativement leurs nombres de pages) les livres de cuisine séparent souvent la recette de la pâte feuilletée des recettes qui l'utilisent. Par exemple, mon livre habituel donne la recette de la pâte feuilletée à la page 39 et la recette de la galette des rois à la page 426. Il ne m'est jamais arrivé de préparer un plat constitué uniquement d'une pâte feuilletée, donc la recette de la pâte feuilletée n'a d'intérêt que si une autre recette y renvoie. Ainsi, lorsque je veux préparer une galette, j'ouvre mon livre à la page 426, la recette qui s'y trouve commence par m'*envoyer* vers la page 39 pour faire la pâte feuilletée[10] et lorsque j'en ai fini avec la recette de la page 39, je *retourne* à la page 426) pour passer à l'étape suivante de la recette de galette.

Cette analogie illustre parfaitement la notion de *procédure* (la recette de la pâte feuilletée) et d'*appel de procédure* (le renvoie de la page 426 vers la page 39, suivi du retour de la page 39 vers l'étape de la page 426 suivant le renvoie).

Ainsi, une procédure est une séquence d'instructions qui n'est exécutée que lorsque l'on en a besoin, que lorsqu'on l'*appelle*.

[9] C'est là l'un des gros avantage de faire les galettes soit même !

[10] Il m'arrive bien de la faire moi-même ! Ce n'est pas si dur que ça… Le raisonnement tient quand on achète la pâte toute prête, en informatique on appelle cela un appel de procédure distant ou « remote procedure call »…

En Orvet on déclare une procédure avec la construction :

**procédure** <nom> **début**

une séquence d'instructions

**fin**

Qui n'a aucun effet autre que déclaratif.

Pour appeler une procédure, on utilise alors tout simplement l'instruction

**appeler** <nom>

où <nom> doit correspondre à une procédure préalablement définie.

Exemple !

```
entier a
entier b
entier t

procédure échanger début

    affecter a à t
    affecter b à a
    affecter t à b

fin

lire a
lire b

montrer a
montrer b

appeler échanger

montrer a
montrer b
```

Dont l'exécution donne :

```
Orvet version 0.1
Chargement du programme
26 lignes chargées
Démarrage de l'exécution

Valeur de l'entier a ? 1
Valeur de l'entier b ? 2
a = 1
b = 2
a = 2
```

```
b = 1

Fin de l'exécution
```

Une exécution avec traces illustre bien ce qui se passe au moment de la définition de la procédure `échanger` et au moment de son appel :

```
Orvet version 0.1
Chargement du programme
26 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier a (initialisé à 0)
5 - Définition de l'entier b (initialisé à 0)
6 - Définition de l'entier t (initialisé à 0)
8 - Définition de la procédure échanger @ 8
Valeur de l'entier a ? 1
16 - Lecture de la valeur de l'entier a
Valeur de l'entier b ? 2
17 - Lecture de la valeur de l'entier b
a = 1
19 - Ecriture de la valeur de l'entier a
b = 2
20 - Ecriture de la valeur de l'entier b
22 - Appel de la procédure échanger
10 - Affectation de la valeur 1 à la variable entière t
11 - Affectation de la valeur 2 à la variable entière a
12 - Affectation de la valeur 1 à la variable entière b
22 - Retour d'appel de la procédure échanger
a = 2
24 - Ecriture de la valeur de l'entier a
b = 1
25 - Ecriture de la valeur de l'entier b

Fin de l'exécution
```

Où, on l'aura compris, les instructions du corps de la procédure sont en vert et celle des instructions de définition et d'appel de la procédure en orange (on remarque également qu'Orvet, en mode trace, trace également le retour d'appel d'une procédure).

Les procédures peuvent tout à fait appeler d'autres procédures[11] :

```
entier a
entier b
entier t

procédure échanger début

    affecter a à t
    affecter b à a
    affecter t à b
```

[11] Elles peuvent même se rappeler elles-mêmes, c'est ce que l'on appelle la récursivité, c'est le sujet de la section 15 !

```
fin

procédure faire_échange début

    appeler échanger

fin

lire a
lire b

montrer a
montrer b

appeler faire_échange

montrer a
montrer b
```

Nous ferons grâce de l'exécution de ce programme puisqu'elle est parfaitement identique à celle du programme précédent (hormis dans le cas d'une exécution en mode trace).

Passons à un exemple plus intéressant.

L'algorithme d'Euclide, pour le calcul du plus grand commun diviseur de deux entiers, est vraisemblablement à la théorie des nombres ce que la pâte feuilletée est à la cuisine[12] : un élément très utile que l'on utilise dans de nombreux contextes.

Cela fait donc sens d'implémenter cet algorithme sous la forme d'une procédure que l'on appellera au besoin, par exemple pour simplifier une fraction :

```
05. entier a
06. entier b
07. entier r
08. entier pgcd
09. entier t
10. booléen échanger
11. booléen continuer
12.
13. $ Calcule le PGCD de a et b dans pgcd à l'aide
14. $ de l'algorithme d'Euclide.
15. procédure plus_grand_commun_diviseur début
16.
17.     minorer b par a dans échanger
18.
19.     si échanger alors
20.
```

[12] Désolé !

```
21.         affecter a à t
22.         affecter b à a
23.         affecter t à b
24.
25.     fin
26.
27.     $ A ce stade, on a a >= b.
28.
29.     affecter vrai à continuer
30.
31.     tant que continuer faire
32.
33.         réduire a modulo b dans r
34.         affecter b à a
35.         affecter r à b
36.
37.         différencier b avec 0 dans continuer
38.
39.     fin
40.
41.     affecter a à pgcd
42.
43. fin
44.
45. entier numérateur
46. entier dénominateur
47.
48. lire numérateur
49. lire dénominateur
50.
51. affecter numérateur à a
52. affecter dénominateur à b
53.
54. appeler plus_grand_commun_diviseur
55.
56. montrer pgcd
57.
58. écrire La fraction $numérateur / $dénominateur ...
59.
60. diviser numérateur par pgcd dans numérateur
61. diviser dénominateur par pgcd dans dénominateur
62.
63. écrire ... Se simplifie en $numérateur / $dénominateur
```

Exemple d'exécution :

```
Orvet version 0.1
Chargement du programme
64 lignes chargées
Démarrage de l'exécution
```

```
Valeur de l'entier numérateur ? 100
Valeur de l'entier dénominateur ? 220
pgcd = 20
La fraction 100 / 220 ...
... Se simplifie en 5 / 11

Fin de l'exécution
```

Suite à l'instruction d'appel de la ligne 54, la procédure `plus_grand_commun_diviseur` commence son exécution à la ligne 15 avec `a` à 100 et `b` à 220. Comme `a` minore `b`, `échanger` prend `vrai` à la ligne 17 et le bloc d'instructions conditionnel suivant (lignes 21, 22 et 23) est exécuté, résultant en l'échange des valeurs de `a` et de `b`. A ce stade `a` vaut 220 et `b` vaut 100. C'est ainsi que l'on démarre l'exécution du corps de la boucle de la ligne 31 : `r` prend 20 à la ligne 33, puis `a` prend 100 (ligne 34) et `b` prend 20 (ligne 35), `b` n'étant pas nul on continue. A l'itération suivante `r` prend cette fois 0 à la ligne 33 (100 est évidemment divisible par 20), `a` prend 20 (ligne 34) et `b` prend 0 (ligne 35) donc `continuer` passe à `faux` et la boucle s'arrête. La variable `pgcd` prend ainsi 20, l'exécution de la procédure se termine, et ce résultat sera exploité dans la suite de l'exécution qui reprend après l'appel de la procédure, donc ligne 55.

Pour illustrer tout cela plus précisément, voici l'exécution précédente en mode trace :

```
Orvet version 0.1
Chargement du programme
64 lignes chargées
Démarrage de l'exécution

5 - Définition de l'entier a (initialisé à 0)
6 - Définition de l'entier b (initialisé à 0)
7 - Définition de l'entier r (initialisé à 0)
8 - Définition de l'entier pgcd (initialisé à 0)
9 - Définition de l'entier t (initialisé à 0)
10 - Définition du booléen échanger (initialisé à faux)
11 - Définition du booléen continuer (initialisé à faux)
15 - Définition de la procédure plus_grand_commun_diviseur @ 15
45 - Définition de l'entier numérateur (initialisé à 0)
46 - Définition de l'entier dénominateur (initialisé à 0)
Valeur de l'entier numérateur ? 100
48 - Lecture de la valeur de l'entier numérateur
Valeur de l'entier dénominateur ? 220
49 - Lecture de la valeur de l'entier dénominateur
51 - Affectation de la valeur 100 à la variable entière a
52 - Affectation de la valeur 220 à la variable entière b
54 - Appel de la procédure plus_grand_commun_diviseur
17 - Minoration de la valeur 220 par la valeur 100 dans la variable booléenne
échanger
19 - Condition sur échanger à vrai
21 - Affectation de la valeur 100 à la variable entière t
22 - Affectation de la valeur 220 à la variable entière a
23 - Affectation de la valeur 100 à la variable entière b
29 - Affectation de la valeur vrai à la variable booléenne continuer
31 - Bouclage sur continuer
33 - Réduction de la valeur 220 modulo la valeur 100 dans la variable entière r
34 - Affectation de la valeur 100 à la variable entière a
35 - Affectation de la valeur 20 à la variable entière b
37 - Test de la non égalité des valeurs 20 et 0 dans la variable booléenne
continuer
```

```
33 - Réduction de la valeur 100 modulo la valeur 20 dans la variable entière r
34 - Affectation de la valeur 20 à la variable entière a
35 - Affectation de la valeur 0 à la variable entière b
37 - Test de la non égalité des valeurs 0 et 0 dans la variable booléenne continuer
41 - Affectation de la valeur 20 à la variable entière pgcd
54 - Retour d'appel de la procédure plus_grand_commun_diviseur
pgcd = 20
56 - Ecriture de la valeur de l'entier pgcd
La fraction 100 / 220 ...
58 - Ecriture ci-dessus
60 - Division de la valeur 100 par la valeur 20 dans la variable entière numérateur
61 - Division de la valeur 220 par la valeur 20 dans la variable entière
dénominateur
... Se simplifie en 5 / 11
63 - Ecriture ci-dessus

Fin de l'exécution
```

Voilà pour les procédures qui complètent ainsi les capacités d'Orvet...

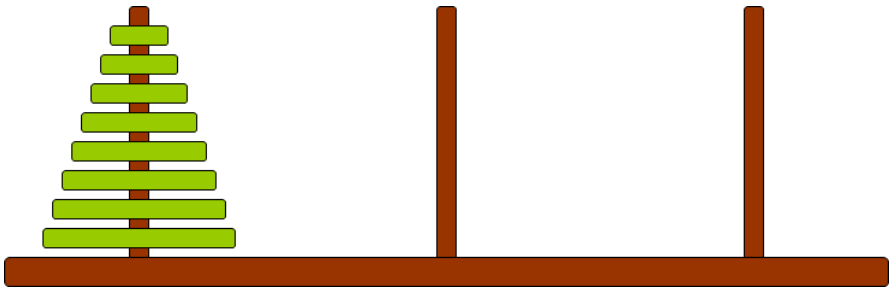
## 15. La récursivité

La récursivité c'est l'art et la manière de résoudre certains problèmes simplement, en faisant l'hypothèse qu'un sous-problème du problème est déjà résolu et ainsi de suite (on résout le sous-problème en faisant l'hypothèse qu'un sous-sous-problème est lui-même déjà résolu) jusqu'à ce que l'on tombe sur un sous-…-sous-problème tellement simple qu'il n'y a quasiment rien à faire, voire rien à faire du tout !

N'en déplaise à l'adage populaire, la récursivité c'est donc un peu l'art de « remettre à plus tard ce que l'on peut faite maintenant » car « plus tard ce sera plus simple ». Bien que cela semble un peu effrayant au début, la récursivité permet bien d'écrire des programmes concis, simples et, lorsque l'on a un peu l'habitude, facile à comprendre.

Pour permettre d'appréhender un peu cette technique, rien de mieux (à mon sens) que le casse-tête des tours Hanoï[13] !

Le jeu est illustré ci-dessous.



[13] Il se trouve que cela fait maintenant plusieurs années que je l'utilise dans le cadre d'ateliers de mathématiques discrètes à l'école maternelle (grande section) et à l'école primaire (http://sirdeyre.free.fr/maths/). J'ai construit une version « géante » du jeu et, en commençant avec deux disques puis en augmentant progressivement le nombre de disques, en 15-20 minutes, un groupe de 6 à 8 enfants arrive à résoudre le jeu jusqu'à 5-6 disques en manipulant quasiment sans aide. Donc, il ne faut pas avoir peur…

Il y a trois piquets, les « tours », que l'on va numéroter de gauche à droite 0, 1 et 2. Au départ, il y a un nombre *n* de disques de diamètres croissants (haut en bas), ici 8 dans le jeu habituel 10, positionnés sur la tour 0. Le but du jeu est de recréer la pile de disques initiale sur la tour 2, en déplaçant les disques un à un et en ne mettant jamais un disque plus grand sur un disque plus petit. Ouf !

Analysons un peu le problème et avant de nous lancer avec 8 ou 10 disques voyons ce qu'il se passe lorsqu'il y a très peu de disques.

Lorsqu'il n'y a aucun disque, eh bien, grande surprise ! Il n'y a rien à faire !

Lorsqu'il n'y a initialement qu'un seul disque sur la tour 0 alors c'est très simple, il suffit de déplacer l'unique disque de la tour 0 vers la tour 2.

Lorsqu'il y a initialement deux disques sur la tour 0, les choses sont à peine plus compliquées : on déplace le petit disque sur la tour 1, puis le grand disque sur la tour 2 et enfin le petit disque de la tour 1 à la tour 2. Tout va bien. Plus généralement, si l'on a une pile de deux disques sur la tour *s* (comme « source ») et que l'on souhaite la déplacer vers la tour *d* (comme « destination ») alors on commence par déplacer le petit disque depuis la tour *s* vers la troisième tour *p* (comme « pivot »), on déplace ensuite le gros disque de la tour *s* vers la tour *d* et, enfin, on déplace de nouveau le petit disque cette fois de la tour *p* vers la tour *d*. Nous venons donc d'énoncer une sorte de procédure qui nous permet de déplacer une pile de deux disques de n'importe quelle tour (*s*) vers n'importe quelle autre tour (*d*) en respectant les règles du jeu.

Et à trois disques ? C'est là que l'on va pouvoir astucieusement utiliser la procédure que nous venons de définir.

Lorsqu'il y a initialement trois disques sur la tour 0, eh bien, remarquons qu'une pile de trois disques ce n'est pas autre chose qu'un gros disque avec une pile de deux disques (donc plus petits) au-dessus. Si l'on était capable de déplacer cette pile de deux disques (toujours en respectant les règles du jeu) sur la tour 1, on pourrait alors déplacer le gros disque sur la tour 2 et si, enfin, on pouvait déplacer de nouveau la pile de deux disques que nous avons mis sur la tour 1 vers la tour 2 (toujours en accord avec les règles) alors le tour serait joué ! Cela tombe bien, nous venons justement de définir comment déplacer une pile de deux disques de n'importe quelle tour vers n'importe quelle autre tour.

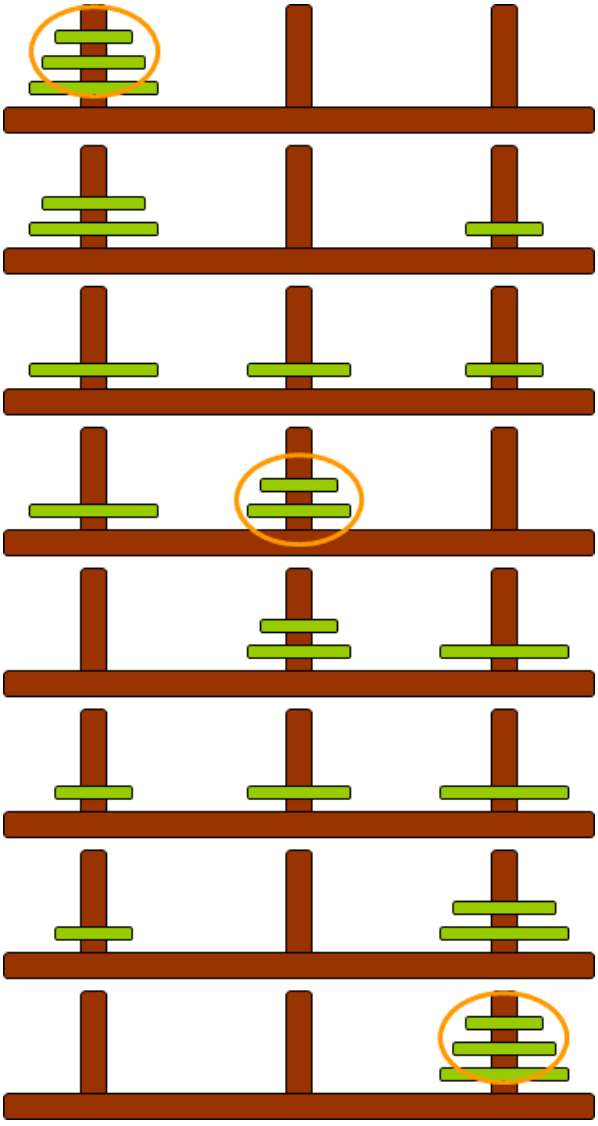
La solution est alors :

1. Déplacer la pile de deux disques de la tour 0 vers la tour 1 (selon la procédure du cas à deux disques avec *s*=0, *d*=1 et *p*=2 donc on déplace le plus petit disque de la tour 0 vers la tour 2, le disque moyen de la tour 0 vers la tour 1, puis on déplace de nouveau le plus petit disque de la tour 2 vers la tour 1) ;
2. Déplacer le plus gros disque (il n'y a plus de disque au-dessus de lui) de la tour 0 vers la tour 2 (il n'y a pas de disque sur la tour 2) ;
3. Déplacer de nouveau la pile de deux disques de la tour 1 vers la tour 2 (selon la procédure du cas à deux disques avec *s*=1, *d*=2 et *p*=0 donc on déplace le plus petit

disque de la tour 1 vers la tour 0, le disque moyen de la tour 1 vers la tour 2, puis on déplace de nouveau le plus petit disque de la tour 0 vers la tour 2).

Et voilà !

Illustration ci-dessous :



Et maintenant, quid du problème à *n* disques ?

Pour déplacer une pile de *n* disques de la tour 0 vers la tour 2 on commence par déplacer (toujours en respectant les règles du jeu) la pile de *n*-1 disques (i.e. le dessus de la pile de *n* disques) de la tour 0 vers la tour 1, puis on déplace le plus gros disque de la tour 0 vers la tour 2 (il n'y a à ce stade aucun disque sur la tour 2), puis on déplace de nouveau la pile de *n*-1 disques (dont tous les disques sont plus petits que le plus gros disque que nous venons de déplacer) vers la tour 2.

Et ainsi de suite avec des piles de disques à chaque fois plus petites (pour déplacer la pile de *n*-1 disques de 0 vers 1, première étape ci-dessus, on doit déplacer la pile de *n*-2 disques de 0

vers 2, puis le disque au sommet de la tour 0 vers la tour 1, puis déplacer de nouveau la pile de *n*-2 disques de 2 vers 1), jusqu'à ce qu'il n'y ait rien à déplacer auquel cas on ne fait rien.

Passons à la programmation de tout cela.

Une fois n'est pas coutume, voici un programme Python :

```
def hanoi(n,src,dst,piv):
 if n>0:
   hanoi(n-1,src,piv,dst)
   print(src,'->',dst)
   hanoi(n-1,piv,dst,src)

def hanoi_top(n):
 hanoi(n,0,2,1)

hanoi_top(8)
```

Nous allons écrire ce programme en Orvet. Ce sera un peu plus long mais pas beaucoup plus.

C'est parti !

```
entier source
entier destination
entier pivot
entier temp
entier nombre_disques
booléen rappel_nécessaire

procédure hanoï début

    différencier nombre_disques avec 0 dans rappel_nécessaire

    si rappel_nécessaire alors

        soustraire 1 à nombre_disques dans nombre_disques

        $ Premier rappel, on échange destination et pivot.
        affecter destination à temp
        affecter pivot à destination
        affecter temp à pivot

        $ Récursivité quand tu nous tiens...
        appeler hanoï

        $ Ceci étant fait, on remet les choses bien en place.
        affecter destination à temp
        affecter pivot à destination
        affecter temp à pivot
```

```
        écrire Déplacer sommet tour $source vers tour $destination

        $ Second rappel, on échange source et pivot.
        affecter source à temp
        affecter pivot à source
        affecter temp à pivot

        $ Recursivité quand tu nous tiens (bis)...
        appeler hanoï

        $ Et, de nouveau, on remet les choses bien en place...
        affecter source à temp
        affecter pivot à source
        affecter temp à pivot

        $ ... Y compris pour le nombre de disques !
        ajouter 1 à nombre_disques dans nombre_disques

    fin

fin

lire nombre_disques

affecter 0 à source
affecter 1 à pivot
affecter 2 à destination

appeler hanoï
```

Vérifions que ce programme fonctionne correctement…

Avec 0 disques (trivial) :

```
Orvet version 0.1
Chargement du programme
83 lignes chargées
Démarrage de l'exécution

Valeur de l'entier nombre_disques ? 0

Fin de l'exécution
```

Avec 1 disque (trop facile) :

```
Orvet version 0.1
Chargement du programme
83 lignes chargées
Démarrage de l'exécution

Valeur de l'entier nombre_disques ? 1
```

```
Déplacer sommet tour 0 vers tour 2

Fin de l'exécution
```

Avec 2 disques (facile) :

```
Orvet version 0.1
Chargement du programme
83 lignes chargées
Démarrage de l'exécution

Valeur de l'entier nombre_disques ? 2
Déplacer sommet tour 0 vers tour 1
Déplacer sommet tour 0 vers tour 2
Déplacer sommet tour 1 vers tour 2

Fin de l'exécution
```

Avec 3 disques (pour de vrai) :

```
Orvet version 0.1
Chargement du programme
83 lignes chargées
Démarrage de l'exécution

Valeur de l'entier nombre_disques ? 3
Déplacer sommet tour 0 vers tour 2
Déplacer sommet tour 0 vers tour 1
Déplacer sommet tour 2 vers tour 1
Déplacer sommet tour 0 vers tour 2
Déplacer sommet tour 1 vers tour 0
Déplacer sommet tour 1 vers tour 2
Déplacer sommet tour 0 vers tour 2

Fin de l'exécution
```

Ce qui reproduit bien les déplacements de l'illustration précédente.

On s'arrêtera là, en laissant le soin au lecteur d'exécuter le programme pour des valeurs plus grandes du nombre de disques ainsi qu'en activant le mode trace, si nécessaire, pour bien comprendre ce qui se passe.

Ainsi se termine donc ce premier contact avec la récursivité...

## 16. Installer et utiliser Orvet

Un programme Orvet doit être un fichier à l'extension `.orv`.

L'interpréteur Orvet (c'est-à-dire le programme qui permet d'exécuter les programmes écrits en Orvet) est un programme écrit en Python3 d'à peu près un kilo-ligne.

Par exemple, si le programme de la section 11 se trouve dans le fichier `exemples/premier.orv`, il sera exécuté à l'aide de la commande suivante :

```
$ python3 orvet.py exemples/premier.orv
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 13
13 est un nombre premier

Fin de l'exécution

Appuyer sur Entrée pour fermer...
```

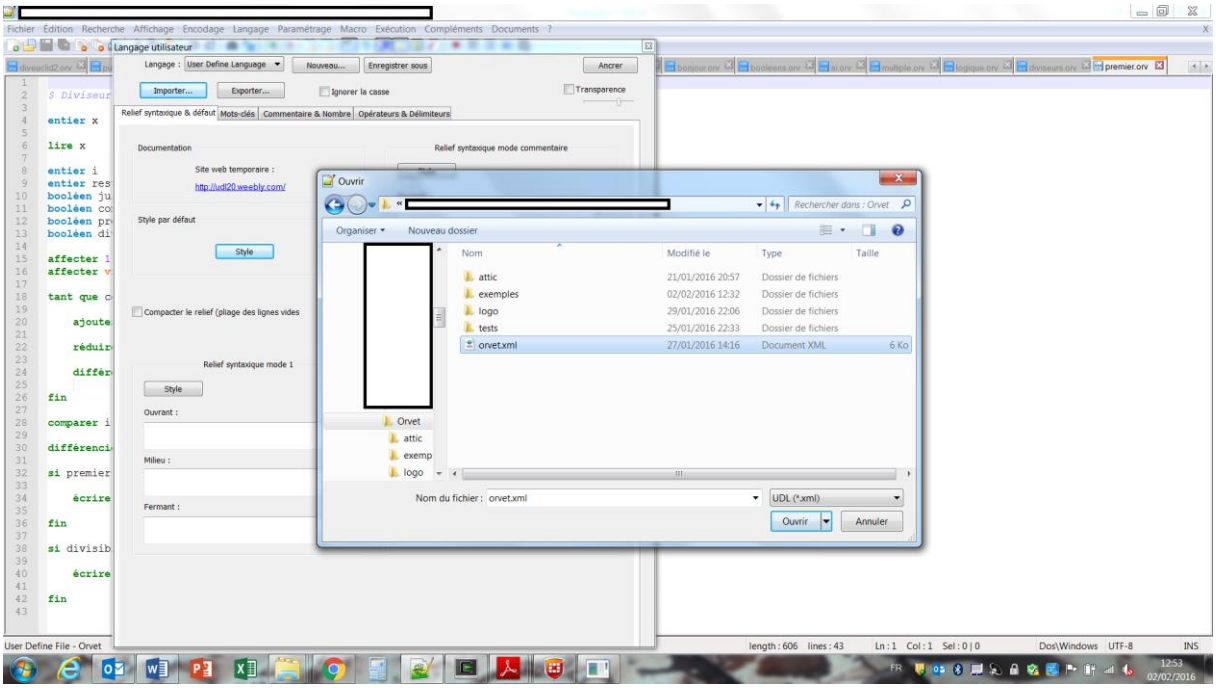
L'option –trace permet d'activer le traçage des instructions tel que nous l'avons fait sections 10 et 11 :

```
$ python3 orvet.py exemples/premier.orv -trace
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

4 - Définition de l'entier x (initialisé à 0)
Valeur de l'entier x ? 13
6 - Lecture de la valeur de l'entier x
8 - Définition de l'entier i (initialisé à 0)
9 - Définition de l'entier reste (initialisé à 0)
10 - Définition du booléen juste (initialisé à faux)
11 - Définition du booléen continuer (initialisé à faux)
12 - Définition du booléen premier (initialisé à faux)
13 - Définition du booléen divisible (initialisé à faux)
15 - Affectation de la valeur 1 à la variable entière i
16 - Affectation de la valeur vrai à la variable booléenne continuer
18 - Bouclage sur continuer

...

28 - Comparaison des valeurs 13 et 13 dans la variable booléenne premier
30 - Test de la non égalité des valeurs 13 et 13 dans la variable booléenne divisible
32 - Condition sur premier à vrai
13 est un nombre premier
34 - Ecriture ci-dessus
38 - Condition sur divisible à faux

Fin de l'exécution
Mémoire :
Entiers :
i = 13
reste = 0
x = 13
Booléens :
continuer = faux
divisible = faux
juste = faux
premier = vrai
```

```
Appuyer sur Entrée pour fermer...
```

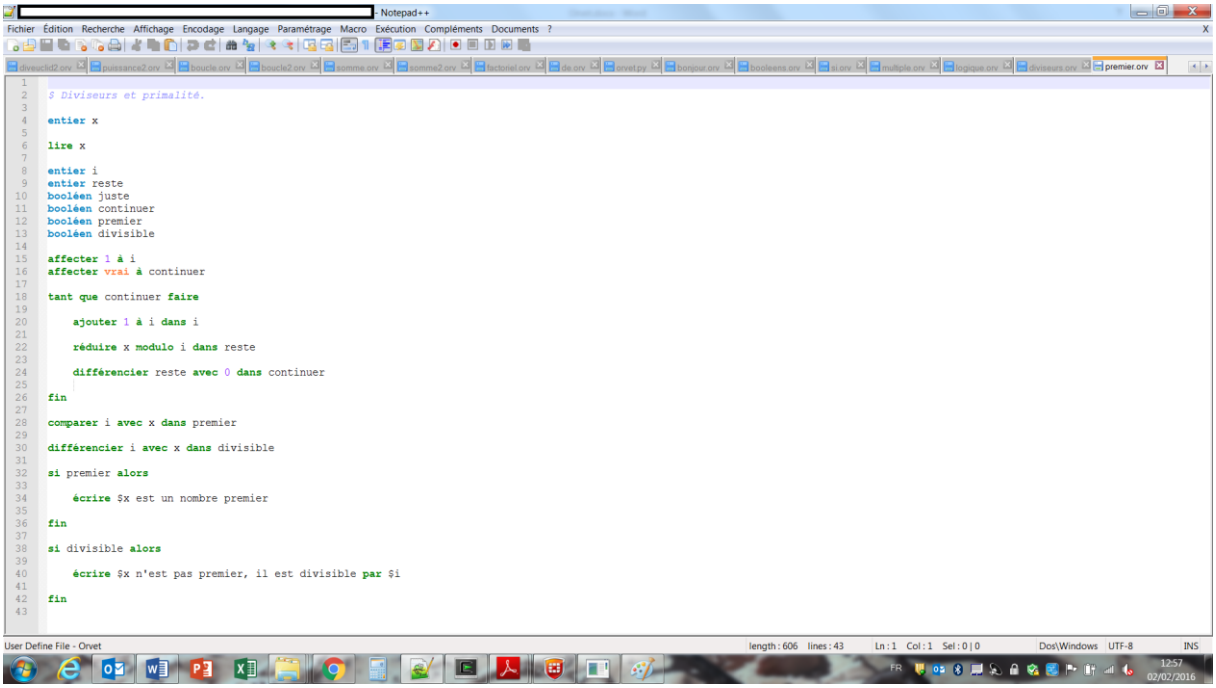
Les traces additionnelles ont été montrées ci-dessus en petit.

Pour rendre les choses plus agréables, Orvet vient avec un fichier `orvet.xml` qui définit un style pour l'éditeur Notepad++ (notepad-plus-plus.org/fr/). Une fois l'éditeur installé, dans le menu « Langage », on sélectionne « Définissez votre langage… » et le bouton « Importer » permet de charger le fichier `orvet.xml` :

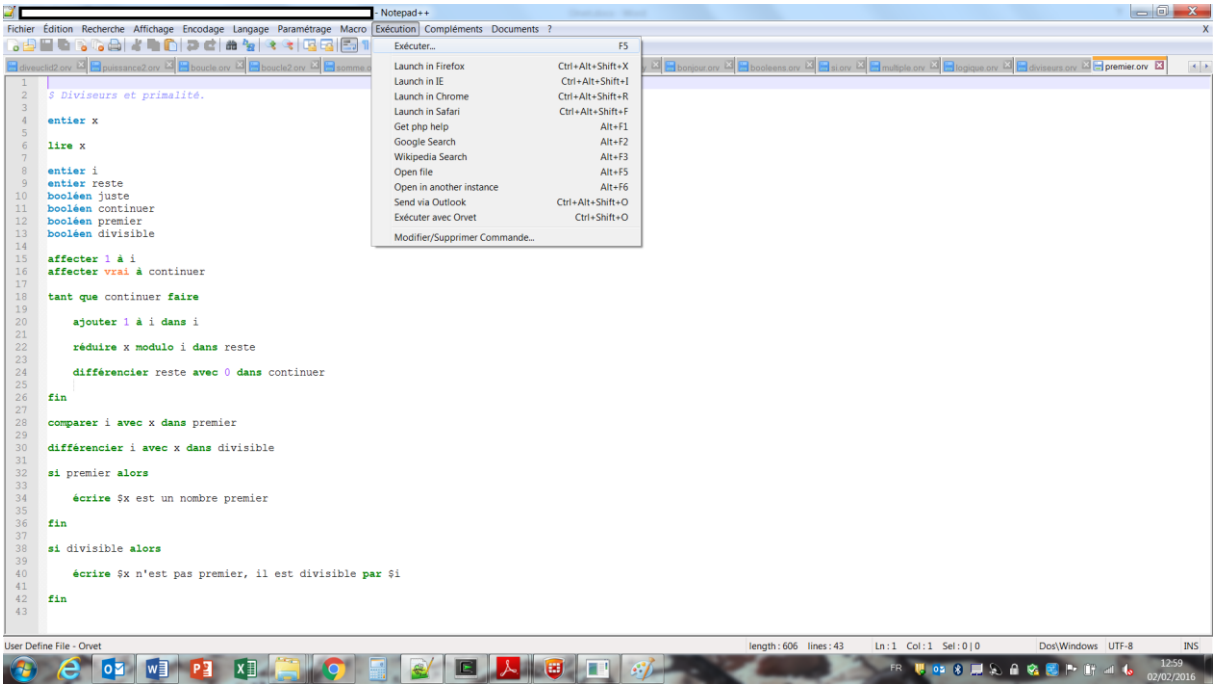



Ainsi, Notepad++ appliquera automatiquement ce style aux fichiers dont l'extension est `.orv`.

Une fois le style installé, à l'écran, cela donne quelque chose comme ça :



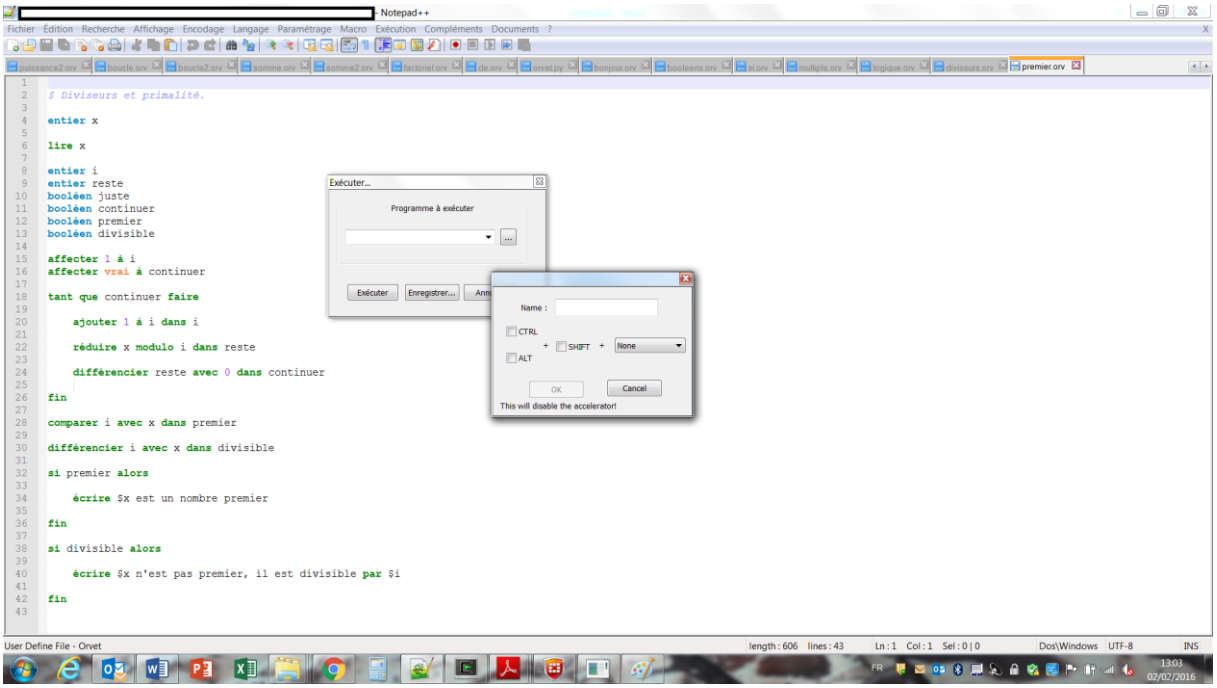

Enfin, dans le menu « Exécution » on peut choisir « Exécuter » :



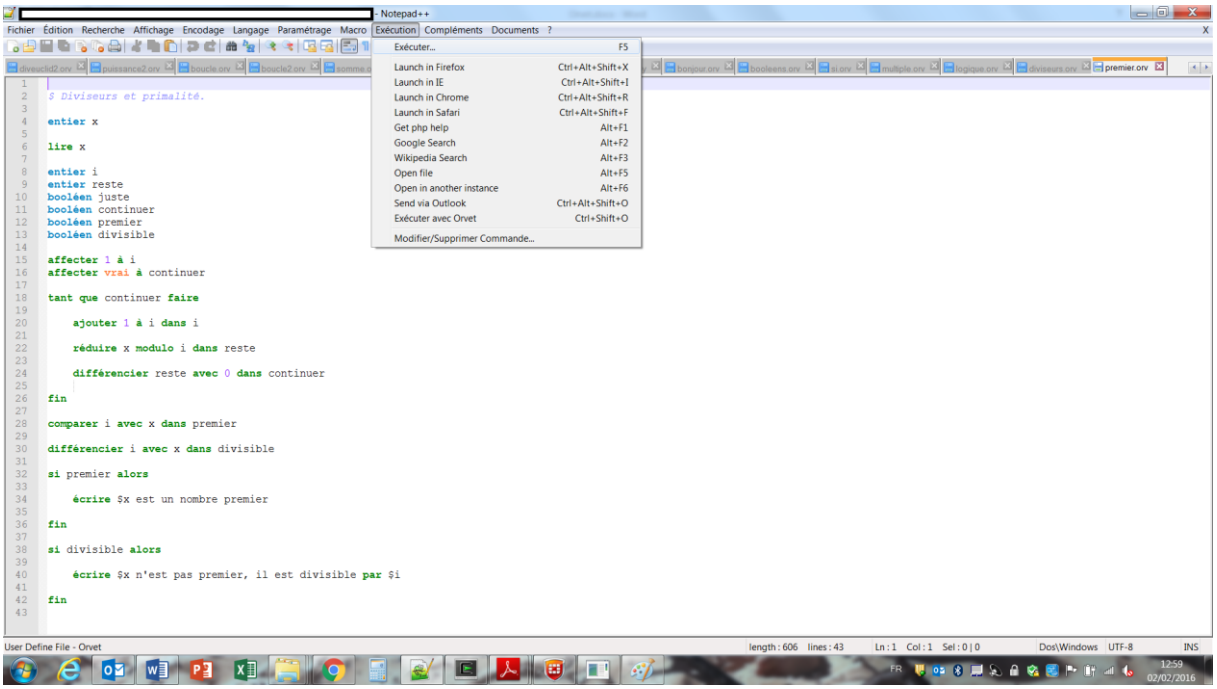

Alors en entrant la commande :

```
C:\...\python3.4m.exe C:\...\orvet.py "$(FULL_CURRENT_PATH)"
```

dans la boîte de dialogue  et en sélectionnant « Enregistrer » :

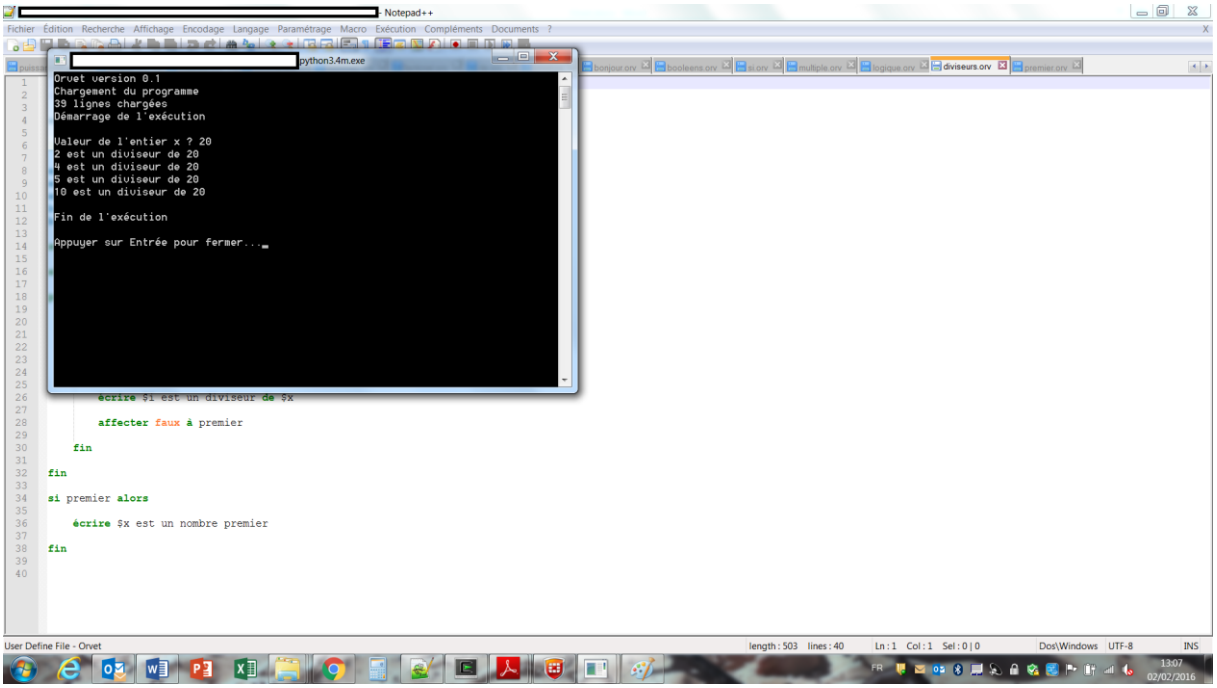



On peut créer une commande et un accélérateur clavier qui permet de lancer l'exécution d'un programme Orvet directement depuis Notepad++ :




Sur ma configuration, la combinaison « Control/Shift/O » permet de lancer cette exécution.

Cela donne ainsi :




Cela permet ainsi à un enfant – c'était notre but initial – de maniper simplement depuis Notepad++ une fois la configuration faite par un adulte !

## 17. Démarrage rapide (en moins d'une minute)

Cette section est pour les utilisateurs (avertis) pressés qui veulent essayer faire rapidement fonctionner Orvet (en ligne de commande).

Dans le répertoire Orvet (extrait de l'archive) :

```
$ python3 orvet.py exemples/bonjour.orv
Orvet version 0.1
Chargement du programme
4 lignes chargées
Démarrage de l'exécution

Bonjour monde !

Fin de l'exécution

Appuyer sur Entrée pour fermer...
```

```
$ python3 orvet.py exemples/premier.orv
Orvet version 0.1
Chargement du programme
42 lignes chargées
Démarrage de l'exécution

Valeur de l'entier x ? 23
23 est un nombre premier

Fin de l'exécution
```

```
Appuyer sur Entrée pour fermer...
$
```

Pour jauger les rudiments du langage, jeter donc un œil à `exemples/bonjour.orv` (« Hello world ») et à `exemples/premier.orv` (un premier programme non complétement trivial).

Lire la section précédente (section 14) pour les détails de l'intégration avec notepad++ (essentiellement pour configurer un système utilisable pour vos enfants).

Lire ce manuel pour les détails du langage (il est censé être court et synthétique et – j'espère en tout cas – accessible à des débutants).

## 18. Quelques subtilités

Cette section est plutôt à l'attention des adultes[14] (voire des adultes informaticiens☺). Je ne prends pas de gants du coup.

Orvet est un langage interprété avec une gestion des variables très simples (sans gestion de « scope ») mais qui peut avoir des effets un peu déroutant, il suffit de le savoir. Une variable est définie une fois que l'instruction de déclaration a été exécutée, pour le reste de l'exécution du programme.

Donc si on déclare une variable dans le corps d'une boucle, par exemple, alors on aura une erreur à la deuxième exécution du corps de boucle (la variable étant considérée comme déjà définie).

Exemple :

```
entier i

pour i de 1 à 10 faire

    booléen b

    montrer b

fin
```

Dont l'exécution donne :

```
Orvet version 0.1
Chargement du programme
12 lignes chargées
Démarrage de l'exécution
```

[14] En espérant qu'une bonne partie de ce qui précède soit accessible par des non-adultes ou des novices de la programmation…

```
b = faux
Erreur ligne 8 : une variable b est déjà définie
Erreur à la ligne 8 : instruction inconnue ou séquence interrompue
Interruption boucle de la ligne 6
Erreur à la ligne 6 : instruction inconnue ou séquence interrompue

Fin de l'exécution
```

*D'une manière générale, il est donc recommandé d'éviter de déclarer des variables à l'intérieur des corps de boucle ou des blocs d'exécution conditionnelle*.

D'une manière générale, également, lorsqu'une erreur de syntaxe ou d'exécution se présente l'interpréteur Orvet indique un message qui se veut aider à comprendre la nature de l'erreur (e.g. également section 4 avec la division par zéro).

De même, si le corps de boucle n'est pas exécuté (cas d'un tant que) alors la variable ne sera pas définie après la boucle car l'instruction de déclaration n'aura pas été exécuté. Même genre de subtilités pour les variables définies dans le corps d'un « si … alors ».

Autre subtilité, l'analyseur syntaxique d'Orvet ne suit pas une grammaire contextuelle. Cela signifie que les mots-clefs du langage ne sont pas des mots-clefs au sens strict. Par exemple, le code suivant marche très bien :

```
booléen booléen

affecter vrai à booléen

montrer booléen
```

Encore une autre subtilité : l'affectation de la variable d'itération d'une boucle « pour … faire » qui n'est effective que jusqu'à la fin de l'exécution du corps de boucle courant (exception faite de la dernière exécution du corps de boucle). Par exemple, le programme suivant :

```
entier i

pour i de 1 à 10 faire

    montrer i

    affecter 100 à i

    montrer i

fin

montrer i
```

Dont l'exécution donne :

```
Orvet version 0.1
Chargement du programme
19 lignes chargées
Démarrage de l'exécution

i = 1
i = 100
i = 2
i = 100
i = 3
i = 100
i = 4
i = 100
i = 5
i = 100
i = 6
i = 100
i = 7
i = 100
i = 8
i = 100
i = 9
i = 100
i = 10
i = 100
i = 100

Fin de l'exécution
```

Il y a quelques autres restrictions/subtilités plus mineures, notamment le fait que l'on ne peut `comparer` ou `différencier` des variables booléennes, ni convertir directement des variables booléennes en entiers. Je rajouterai le support pour ces instructions si j'en ressens le besoin en codant des algorithmes plus compliqués.

## 19. Lexique des instructions d'Orvet

- **`entier`** <var> (entier).
- **`booléen`** <var> (booléen). **`vrai`** et **`faux`** pour les valeurs de vérité.
- **`lire`** <var> (entier seulement).
- **`montrer`** <var> (entier ou booléen).
- **`affecter`** <$var_1$|$val_1$> **`dans`** <var> (entier → entier ou booléen → booléen).
- **`ajouter`** <$var_1$|$val_1$> **`à`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).
- **`multiplier`** <$var_1$|$val_1$> **`par`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).
- **`soustraire`** <$var_2$|$val_2$> **`à`** <$var_1$|$val_1$> **`dans`** <var> (entier × entier → entier).
- **`diviser`** <$var_1$|$val_1$> **`par`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).
- **`réduire`** <$var_1$|$val_1$> **`modulo`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).
- **`élever`** <$var_1$|$val_1$> **`à la puissance`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).
- **`maximiser`** <$var_1$|$val_1$> **`et`** <$var_2$|$val_2$> **`dans`** <var> (entier × entier → entier).

- **minimiser** <var$_1$|val$_1$> **et** <var$_2$|val$_2$> **dans** <var> (entier × entier → entier).
- **tirer** <var> **au hasard entre** <var$_1$|val$_1$> **et** <var$_2$|val$_2$> (entier ← entier × entier).
- **écrire** <mot$_1$|symbole$_1$|**$**var$_1$> <mot$_1$|symbole$_2$|**$**var$_1$> ...
- **comparer** <var$_1$|val$_1$> **avec** <var$_2$|val$_2$> **dans** <var> (entier × entier → booléen).
- **différencier** <var$_1$|val$_1$> **avec** <var$_2$|val$_2$> **dans** <var> (entier × entier → booléen).
- **minorer** <var$_1$|val$_1$> **par** <var$_2$|val$_2$> **dans** <var> (entier × entier → booléen).
- **majorer** <var$_1$|val$_1$> **par** <var$_2$|val$_2$> **dans** <var> (entier × entier → booléen).
- **disjoindre** <var$_1$|val$_1$> **et** <var$_2$|val$_2$> **dans** <var> (booléen × booléen → booléen) – OU logique.
- **conjoindre** <var$_1$|val$_1$> **et** <var$_2$|val$_2$> **dans** <var> (booléen × booléen → booléen) – ET logique.
- **complémenter** <var$_1$|val$_1$> **dans** <var> (booléen → booléen) – NON logique.
- **pour** <var> **de** <var$_1$|val$_1$> **à** <var$_2$|val$_2$> **faire** (entier × entier × entier).
- **tant que** <var> **faire** (booléen).
- **si** <var> **faire** (booléen).
- **fin**.
- **vérifier** <var> (booléen).
- **arrêter**.
- **procédure** <nom> **début** (identifiant).
- **appeler** <nom> (identifiant).

## 20. Extensions futures

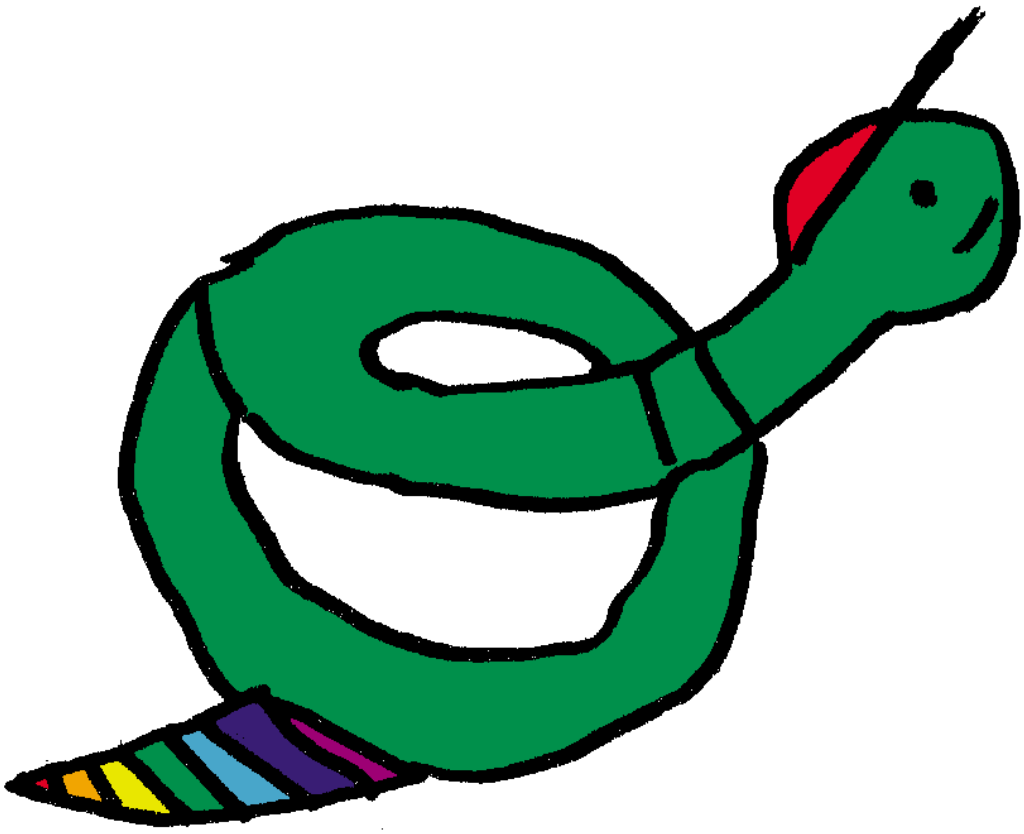
Quelques extensions sont envisagées pour un futur (pas vraiment déterminé pour l'instant mais vraisemblablement pas trop lointain) :

- Une structure conditionnelle « `sinon` <var> `alors` » ?
- ~~Procédures~~ (vraisemblablement associées à une pile globale sur laquelle on pourra `empiler` et `dépiler` des « choses »).
- Ensembles d'entiers (et de booléens ?).
- Tableaux (et listes ?).
- Fractions (le plus simple sera de les gérer sous la forme d'un jeton insécable 1/2, 100/103, etc. pour ce qui est des valeurs immédiates, cela du coup ne dérangera pas la forme syntaxique des instructions).

Dans tous les cas, tout ou une partie de ces instructions seront supportés dans une version ultérieure du langage (V>1 puisqu'on en est à la V0 dans ce guide).

# Aide-mémoire Orvet (version 0)

- **entier** <var> (entier).
- **booléen** <var> (booléen). **vrai** et **faux** pour les valeurs de vérité.
- **lire** <var> (entier seulement).
- **montrer** <var> (entier ou booléen).
- **affecter** <$var_1$|$val_1$> **dans** <var> (entier → entier ou booléen → booléen).
- **ajouter** <$var_1$|$val_1$> **à** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **multiplier** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **soustraire** <$var_2$|$val_2$> **à** <$var_1$|$val_1$> **dans** <var> (entier × entier → entier).
- **diviser** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **réduire** <$var_1$|$val_1$> **modulo** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **élever** <$var_1$|$val_1$> **à la puissance** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **maximiser** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **minimiser** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> (entier × entier → entier).
- **tirer** <var> **au hasard entre** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> (entier ← entier × entier).
- **écrire** <$mot_1$|$symbole_1$|**$**$var_1$> <$mot_1$|$symbole_2$|**$**$var_1$> ...
- **comparer** <$var_1$|$val_1$> **avec** <$var_2$|$val_2$> **dans** <var> (entier × entier → booléen).
- **différencier** <$var_1$|$val_1$> **avec** <$var_2$|$val_2$> **dans** <var> (entier × entier → booléen).
- **minorer** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> (entier × entier → booléen).
- **majorer** <$var_1$|$val_1$> **par** <$var_2$|$val_2$> **dans** <var> (entier × entier → booléen).
- **disjoindre** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> (booléen × booléen → booléen) – OU logique.
- **conjoindre** <$var_1$|$val_1$> **et** <$var_2$|$val_2$> **dans** <var> (booléen × booléen → booléen) – ET logique.
- **complémenter** <$var_1$|$val_1$> **dans** <var> (booléen → booléen) – NON logique.
- **pour** <var> **de** <$var_1$|$val_1$> **à** <$var_2$|$val_2$> **faire** (entier × entier × entier).
- **tant que** <var> **faire** (booléen).
- **si** <var> **faire** (booléen).
- **fin**.
- **vérifier** <var> (booléen).
- **arrêter**.
- **procédure** <nom> **début** (identifiant).
- **appeler** <nom> (identifiant).



« Le petit orvet s’appelle Julien ! » (ainsi que l’a nommé Clément).

FIN DU DOCUMENT