

Programmer avec Orvet

(version 0)



Renaud Sirdey
renaud.sirdey@gmail.com

5 février 2016

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

L. Kronecker

Table des matières

1. Introduction	7
2. Un premier programme	7
3. Déclaration et affectation de variables entières	8
4. Faire des calculs	9
5. Une première structure de boucle	13
6. Tirer au hasard	17
7. Amélioration de l’affichage.....	19
8. Déclarer et affecter des variables booléennes	20
9. Comparer des entiers.....	21
10. Et si ?	23
11. Boucle « tant que »	30
12. Opérateurs logiques	35
13. Arrêt et vérification de conditions	37
14. Installer et utiliser Orvet	39
15. Démarrage rapide (en moins d’une minute).....	43
16. Quelques subtilités.....	44
17. Lexique des instructions d’Orvet.....	46
18. Extensions futures.....	47

1. Introduction

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

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 15, page 43.

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** <var₁|val₁> **à** <var₂|val₂> **dans** <var> ;
- **multiplier** <var₁|val₁> **par** <var₂|val₂> **dans** <var> ;
- **diviser** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (<var₂|val₂> devant être différent de 0) ;
- **réduire** <var₁|val₁> **modulo** <var₂|val₂> **dans** <var> (<var₂|val₂> devant également être différent de 0) ;
- **élever** <var₁|val₁> **à la puissance** <var₂|val₂> **dans** <var> (<var₂|val₂> 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** <var₁|val₁> **et** <var₂|val₂> **dans** <var> ;
- **minimiser** <var₁|val₁> **et** <var₂|val₂> **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 <var1|val1> à <var2|val2> 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* <var₂|val₂>- <var₁|val₁>+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 augmenter².

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 $S = N \times (N+1) / 2$. 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 $S_1=1 \times (1+1)/2=1$, la formule fonctionne. Pour $N=2$ on a $S_2=2 \times (2+1)/2=3=1+2$, tout va bien. Supposons maintenant que la formule fonctionne également pour $N-1$, c'est-à-dire que $S_{N-1}=(N-1) \times (N-1+1)/2=(N-1)N/2$. 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.

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
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 = 3856204823625804217356770659234636406174931095902235902788284
03276373402575165543560686168588507361534030051833058916347592172932
26249885776611495524503935776003464470927924769249558528000000000000
00000000000000000000

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 hasard³.

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

tirer <var> au hasard entre <var₁|val₁> et <var₂|val₂>

Ainsi <var> se retrouvera affectée avec une quelconque des <var₂|val₂>-<var₁|val₁>+1 valeurs entre <var₂|val₂> et <var₁|val₁> (inclusivement, donc). Chacune de ces valeurs possède des chances identiques de sortir (on dit qu'elles sont équiprobables).

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

³ 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 intéressants en informatique, mais là je digresse quelque peu...

```

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 valeurs⁴ :

```

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

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

```

⁴ 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 = 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 limites⁵.

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 influencer 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
```

```
montrer a
```

```
booléen b
```

⁵ Autres que celles de la calculabilité. Non je ne digresse pas plus ici...

```
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 <var₁|val₁> **avec** <var₂|val₂> **dans** <var>

mettra `vrai` dans <var> si la valeur entière <var₁|val₁> est égale à la valeur entière <var₂|val₂>, et `faux` sinon.

Quant à elle, l'instruction

différencier <var₁|val₁> **avec** <var₂|val₂> **dans** <var>

fera le contraire, elle mettra `vrai` dans <var> si la valeur entière <var₁|val₁> est différente de la valeur entière <var₂|val₂>, 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** <var₁|val₁> **par** <var₂|val₂> **dans** <var>.
- **majorer** <var₁|val₁> **par** <var₂|val₂> **dans** <var>.

permettent de tester si <var₂|val₂> est inférieur (strictement) à <var₁|val₁>, pour la première, et si <var₂|val₂> est supérieur (strictement) à <var₁|val₁>, pour la seconde.

10. Et si ?

Nous allons maintenant voir comment ces booléens peuvent influencer 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 entre 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 faire⁶).

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
```

⁶ 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 23⁷. 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 sont 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)
```

⁷ 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.


```

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** `<var1|val1> dans <var>` (booléen → booléen) – NON logique.
- **disjoindre** `<var1|val1> et <var2|val2> dans <var>` (booléen × booléen → booléen) – OU logique.
- **conjoindre** `<var1|val1> et <var2|val2> 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 `<var1|val1>` vaut **vrai** alors `<var>` vaudra **faux** et si `<var1|val1>` vaut **faux** alors `<var>` vaudra **vrai**.

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

<code><var₁ val₁></code>	<code><var></code>
faux	vrai
vrai	faux

L'instruction **disjoindre**, quant à elle, implémente un OU-logique : `<var>` ne vaudra **faux** que si `<var1|val1>` et `<var2|val2>` valent tous les deux **faux**, sinon `<var>` vaudra **vrai**. Sa table de vérité donne donc :

<code><var₁ val₁></code>	<code><var₂ val₂></code>	<code><var></code>
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 `<var1|val1>` et `<var2|val2>` 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. 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

```

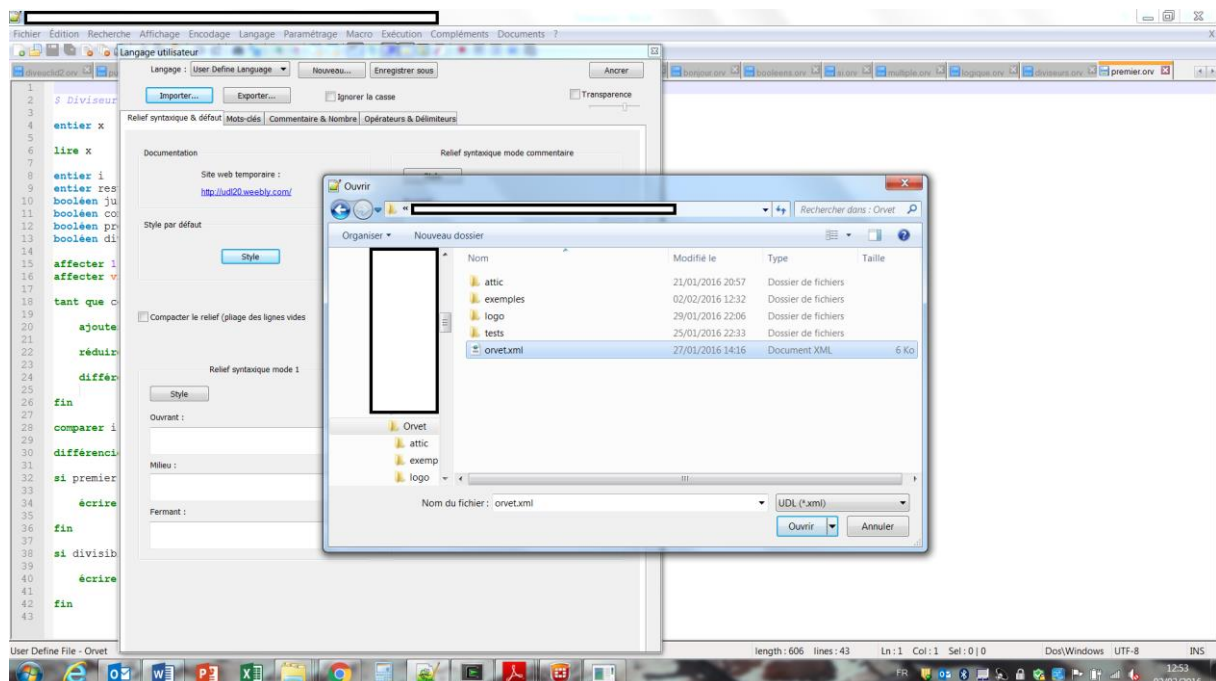
Mémoire :
Entiers :
i = 13
reste = 0
x = 13
Booléens :
continuer = faux
divisible = faux
juste = faux
premier = vrai

```

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` :

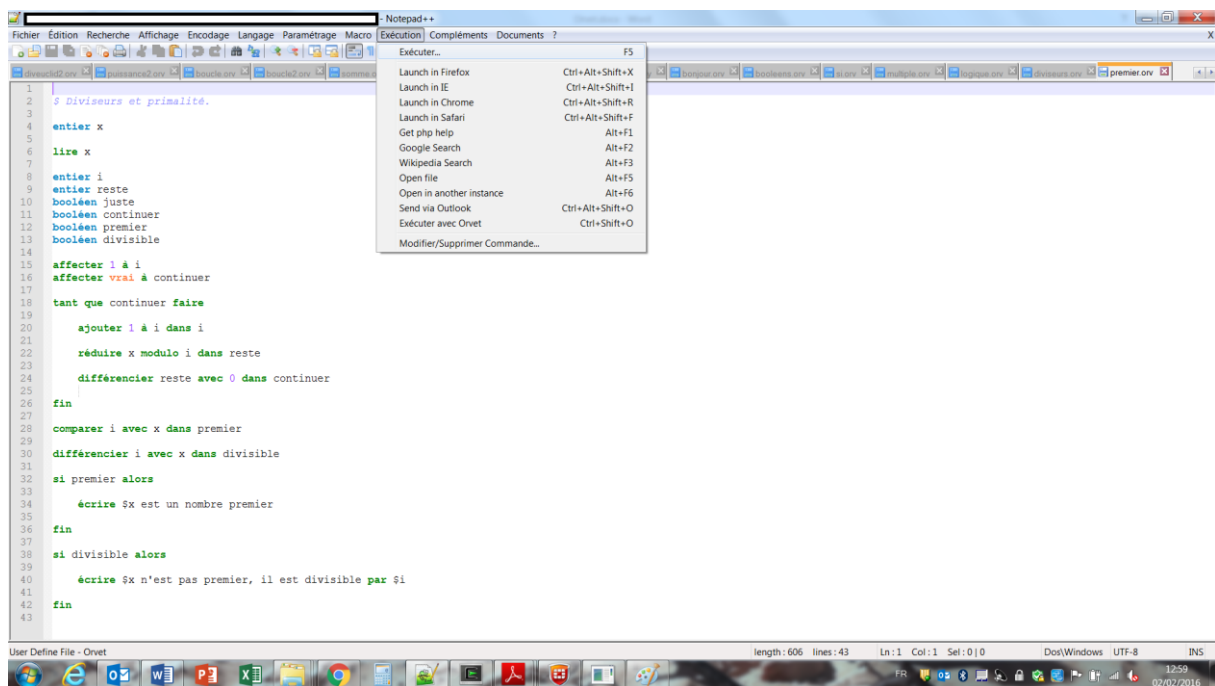


Ainsi, Notepad++ appliquera automatiquement ce style aux fichiers dont l'extension est `.orv`.

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


```
1  $ Diviseurs et primalité.
2
3
4  entier x
5
6  lire x
7
8  entier i
9  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 différencier i avec x dans divisible
30
31 si premier alors
32     écrire $x est un nombre premier
33
34 fin
35
36 si divisible alors
37     écrire $x n'est pas premier, il est divisible par $i
38
39 fin
40
41
42
43
```

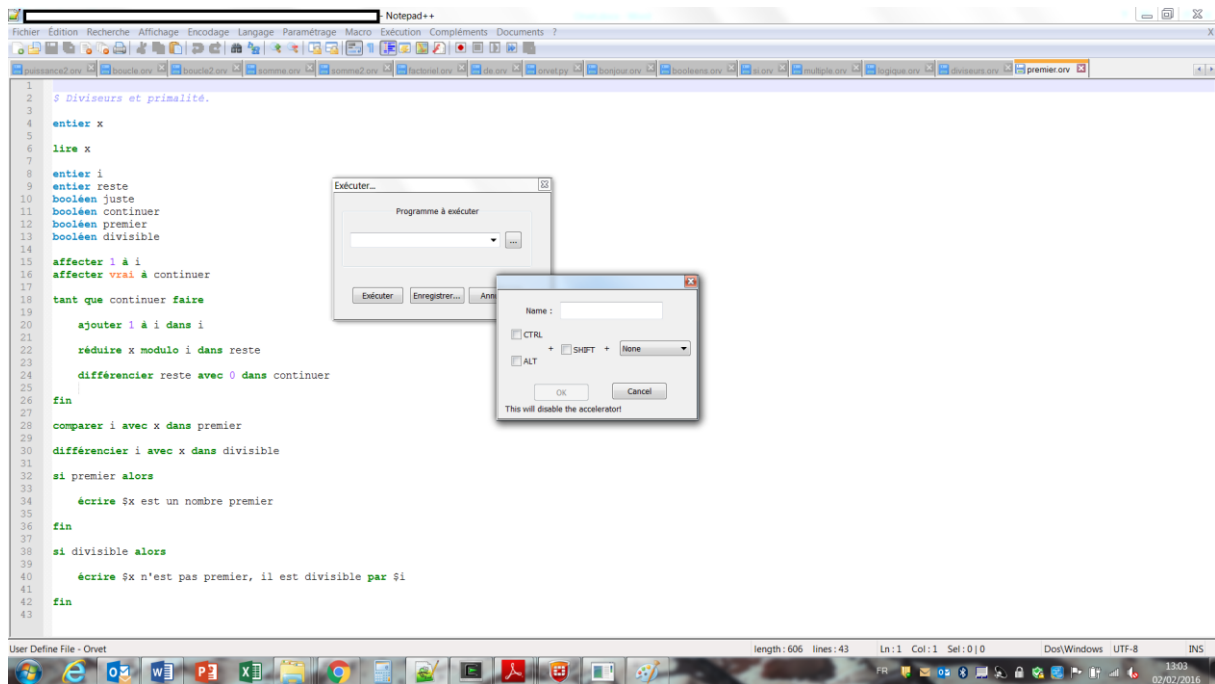
Enfin, dans le menu « Exécution » on peut choisir « Exécuter » :



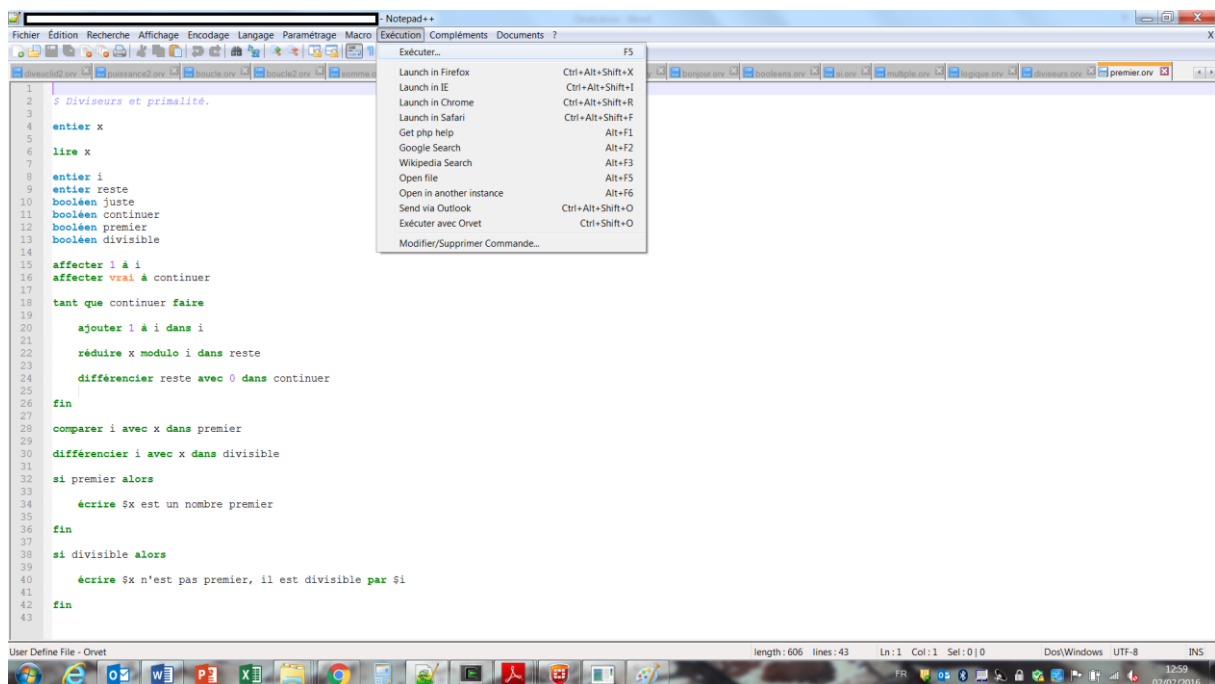
Alors en entrant la commande :

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

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

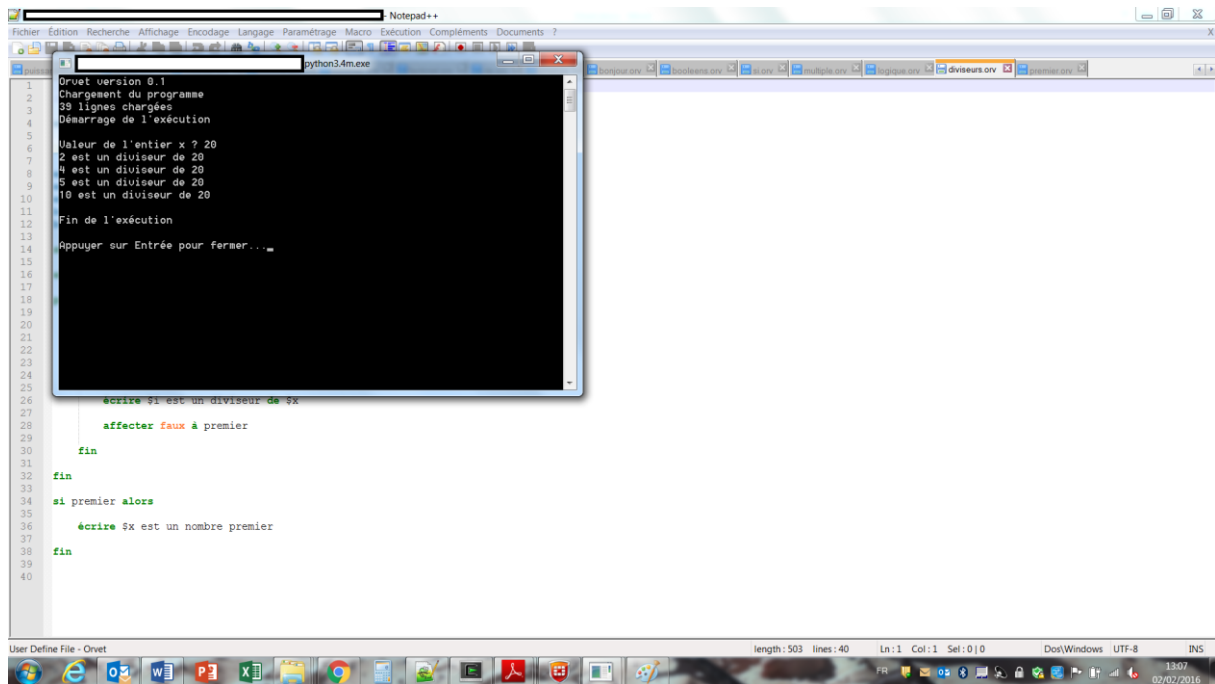


On peut créer une commande et un accélérateur clavier qui permet de lancer l'exécution d'un programme Orvet directement depuis Notepad++ :



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 manipuler simplement depuis Notepad++ une fois la configuration faite par un adulte !

15. 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.ory
```

```
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.ory
```

```
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 complètement 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).

16. Quelques subtilités

Cette section est plutôt à l'attention des adultes⁸ (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
```

⁸ 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 entrée. Je rajouterai le support pour ces instructions si j'en ressens le besoin en codant des algorithmes plus compliqués.

17. 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** <var₁|val₁> **dans** <var> (entier → entier ou booléen → booléen).
- **ajouter** <var₁|val₁> **à** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **multiplier** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **soustraire** <var₂|val₂> **à** <var₁|val₁> **dans** <var> (entier × entier → entier).
- **diviser** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **réduire** <var₁|val₁> **modulo** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **élever** <var₁|val₁> **à la puissance** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **maximiser** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (entier × entier → entier).

- **minimiser** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **tirer** <var> **au hasard entre** <var₁|val₁> **et** <var₂|val₂> (entier ← entier × entier).
- **écrire** <mot₁|symbole₁|\$var₁> <mot₁|symbole₂|\$var₁> ...
- **comparer** <var₁|val₁> **avec** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **différencier** <var₁|val₁> **avec** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **minorer** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **majorer** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **disjoindre** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (booléen × booléen → booléen) – OU logique.
- **conjoindre** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (booléen × booléen → booléen) – ET logique.
- **complémenter** <var₁|val₁> **dans** <var> (booléen → booléen) – NON logique.
- **pour** <var> **de** <var₁|val₁> **à** <var₂|val₂> **faire** (entier × entier × entier).
- **tant que** <var> **faire** (booléen).
- **si** <var> **faire** (booléen).
- **fin.**
- **vérifier** <var> (booléen).
- **arrêter.**

18. Extensions futures

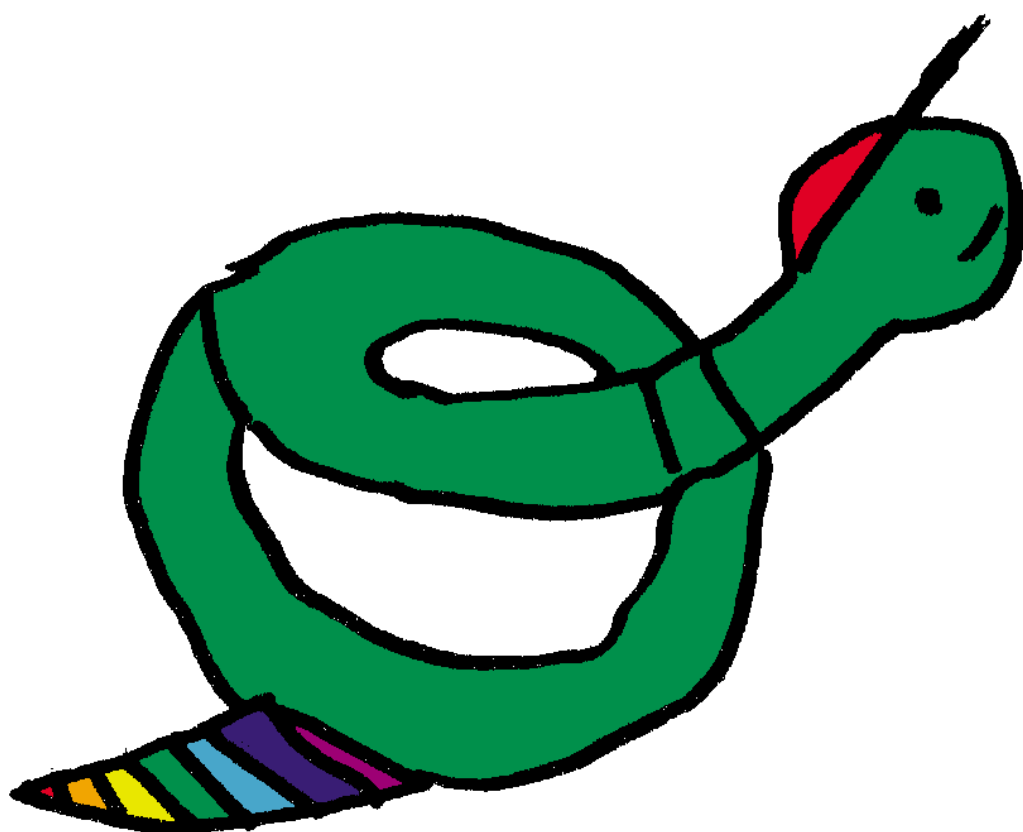
Quelques extensions sont envisagées pour un futur (pas vraiment déterminé pour l’instant mais vraisemblablement pas trop lointain) :

- 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** <var₁|val₁> **dans** <var> (entier → entier ou booléen → booléen).
- **ajouter** <var₁|val₁> **à** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **multiplier** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **soustraire** <var₂|val₂> **à** <var₁|val₁> **dans** <var> (entier × entier → entier).
- **diviser** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **réduire** <var₁|val₁> **modulo** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **élever** <var₁|val₁> **à la puissance** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **maximiser** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **minimiser** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (entier × entier → entier).
- **tirer** <var> **au hasard entre** <var₁|val₁> **et** <var₂|val₂> (entier ← entier × entier).
- **écrire** <mot₁|symbole₁|\$var₁> <mot₁|symbole₂|\$var₁> ...
- **comparer** <var₁|val₁> **avec** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **différencier** <var₁|val₁> **avec** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **minorer** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **majorer** <var₁|val₁> **par** <var₂|val₂> **dans** <var> (entier × entier → booléen).
- **disjoindre** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (booléen × booléen → booléen) – OU logique.
- **conjoindre** <var₁|val₁> **et** <var₂|val₂> **dans** <var> (booléen × booléen → booléen) – ET logique.
- **complémenter** <var₁|val₁> **dans** <var> (booléen → booléen) – NON logique.
- **pour** <var> **de** <var₁|val₁> **à** <var₂|val₂> **faire** (entier × entier × entier).
- **tant que** <var> **faire** (booléen).
- **si** <var> **faire** (booléen).
- **fin**.
- **vérifier** <var> (booléen).
- **arrêter**.



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

FIN DU DOCUMENT