

Séquence 3 - Programmation impérative

Dans cette séquence, nous allons détailler les éléments de base des langages C et OCaml en nous appuyant sur vos connaissances en Python (et aussi, parfois, en les déconstruisant). N'oubliez jamais que Python est un langage de très haut niveau : il permet très rapidement de réaliser des petits programmes, mais cette facilité d'apprentissage a un prix : beaucoup de subtilités de programmation sont masquées et prises en charge par l'interpréteur Python. Nous allons maintenant revenir un peu plus bas pour redécouvrir ces aspects et vous permettre de vous forger une représentation simplifiée mais fiable de ce qui se passe au niveau de la machine.

I. Notre exemple d'étude : l'algorithme d'Euclide

Nous allons reprendre l'algorithme d'Euclide vu dans la séquence précédente et l'implémenter dans les trois langages : C, Python et OCaml.

I. 1. Rappel de l'algorithme d'Euclide

Nous redonnons ici l'écriture en pseudo-code de cet algorithme :

Algorithme 1 : euclide

Donnée : **m**, entier

Donnée : **n**, entier, $m > n$

Variable de travail : **c**, entier

Variable de travail : **d**, entier

Variable de travail : **r**, entier

- | | | |
|---|--|-----------------------|
| 1 | Stocker la valeur de m dans c | $c \leftarrow m$ |
| 2 | Stocker la valeur de n dans d | $d \leftarrow n$ |
| 3 | Stocker le reste de la division euclidienne de c par d dans r | $r \leftarrow c \% d$ |
| 4 | Tant que la valeur contenue dans r est différente de 0 faire | |
| 5 | Stocker la valeur de d dans c | $c \leftarrow d$ |
| 6 | Stocker la valeur de r dans d | $d \leftarrow r$ |
| 7 | Stocker le reste de la division euclidienne de c par d dans r | $r \leftarrow c \% d$ |
| 8 | Renvoyer en sortie de l'algorithme la valeur contenue dans la zone de stockage d | |
-

I. 2. Écriture en trois langages

La Figure 1 détaille la traduction de l'algorithme d'Euclide dans les trois langages à l'aide de l'éditeur de texte **emacs**.

La figure ci-dessous détaille quant à elle les lignes de commandes nécessaires pour faire exécuter ces trois codes par le microprocesseur. Bien entendu, les 3 résultats d'exécution

sont identiques puisque les trois codes décrivent le même algorithme en trois langages différents.

```
golivier@localhost:~/doc/cours/seq2> gcc euclide.c -o euclide_exe
golivier@localhost:~/doc/cours/seq2> ./euclide_exe
Le PGCD de 56 et 16 est 8
golivier@localhost:~/doc/cours/seq2> python3 ./euclide.py
Le PGCD de 56 et 16 est 8
golivier@localhost:~/doc/cours/seq2> ocaml ./euclide.ml
Le PGCD de 56 et 16 est 8
golivier@localhost:~/doc/cours/seq2> █
```

Compilation du code C puis exécution

Interprétation du code Python

Interprétation du code OCaml

Afin d'initier l'apprentissage des langages C et OCaml, nous nous proposons de comparer ces 3 traductions de l'algorithme d'Euclide. Il s'agit de repérer les ponts entre ces trois langages mais aussi les différences (lexicales, syntaxiques, conceptuelles...)

II. Identificateurs

Définition 1 (Identificateur)

Un **identificateur** est un nom, choisi par le programmeur, pour désigner une entité d'un programme : nom d'une variable, nom d'une fonction, nom d'une constante symbolique, nom d'un type défini.

Pour être valide, il doit respecter certaines contraintes syntaxiques :

- Un identificateur ne peut pas contenir d'espace ou de caractères blancs comme la tabulation. En effet, s'il contenait un espace, le compilateur ou l'interpréteur ne peut savoir s'il s'agit d'un seul ou de deux identificateurs !
- Seules les lettres minuscules ou majuscules sans accent, les chiffres et le tiret du bas `_` (ou tiret du 8, appelé aussi *underscore*) peuvent former un identificateur.
- Les signes diacritiques¹ sont interdits.
- En outre, le premier caractère est toujours une lettre.

En théorie, toutes les combinaisons de caractères autorisés sont possibles. Mais l'usage a consacré certaines pratiques que nous adoptons dans ce cours.

- Les noms des variables et des fonctions ne comportent pas de majuscules : `var1` plutôt que `Var1`
- Les constantes symboliques ne contiennent pas de minuscules : `PI`.
- Le tiret du bas `_` ou tiret du 8, aussi appelé *underscore*, est utilisé pour séparer les mots : `un_exemple_de_nom`

Il convient également de choisir le nom d'un identificateur pour qu'il fasse sens. La lisibilité du code source en est ainsi améliorée, pour vous même, et aussi pour vos collaborateurs et relecteurs.

Tous les mots ne peuvent cependant pas être des identificateurs. C'est en particulier le cas les **mots clés** du langage. Ces derniers ont une sémantique précise qui les associe à une action particulière dans le cadre du langage. Les Figures 2, 3 et 4 donnent l'ensemble des mots clés des trois langages C, Python et OCaml.

1. accents, trémas, tildes, etc.

III. Commentaires et documentation de code

Les commentaires sont ajoutés par le programmeur pour faciliter la compréhension du code (pour lui même ou pour un autre programmeur). Ils sont facultatifs et n'ont aucune influence sur l'exécution : ils sont de toute façon nettoyés lors de la phase de pré-traitement avant la compilation ou l'interprétation (cf Séquence 2). Néanmoins, ils sont très fortement conseillés et appréciés des autres programmeurs et relecteurs.

III. 1. Commentaires en langage C

En langage C, les commentaires peuvent s'écrire de deux manières :

- en utilisant `//` : tous les caractères écrits après ce symbole sont des commentaires, jusqu'à la fin de la ligne
- en utilisant `/* commentaires du programmeur */` : tous les caractères entre `/*` et `*/` sont considérés comme des commentaires

```
1
2 /** Commentaire long sur plusieurs lignes.
3     Utilisé par l'outil de documentation automatique C/C++ Doxygen
4
5     @file commentaires.c
6     @brief Fichier exemple pour montrer les commentaires et la doc auto
7     @author G. OLIVIER
8     @version 1.0.0
9     @date 01/09/2022
10 */
11
12 /**
13  * @brief Fonction bête qui ne sert à rien.
14  * @param n: un entier qu'on additionnera
15  * @return retourne 0 si tout va bien, ou un code d'erreur entier différent
16  * de 0 sinon.
17 */
18 int fonction_bete(int n)
19 {
20     int a = 0; // ceci est un commentaire court simple
21     a = a + n; /** commentaire court pris en compte par la doc auto*/
22
23     /* Ici je peux écrire des commentaires non pris en compte par la doc
24        sur plusieurs lignes
25        .....
26     */
27     return 0;
28 }
```

III. 2. Commentaires en langage Python

En langage Python, les commentaires peuvent s'écrire de deux manières :

- en utilisant `#` : tous les caractères écrits après ce symbole sont des commentaires, jusqu'à la fin de la ligne ;

- en utilisant trois guillemets ouvrants et trois guillemets fermants : tous les caractères entre `'''` et `'''` sont considérés comme des commentaires.

```
1
2 """ @package Commentaire long sur plusieurs lignes.
3     Utilisé par l'outil de documentation automatique Doxygen, pydoc3...etc
4
5 @file commentaires.c
6 @brief Fichier exemple pour montrer les commentaires et la doc auto
7 @author G. OLIVIER
8 @version 1.0.0
9 @date 01/09/2022
10 """
11
12
13
14 def fonction_bete(n):
15     """
16     @brief Fonction bête qui ne sert à rien
17     @param n: un entier qu'on additionnera
18     @return: retourne 0 si tout va bien, ou un code d'erreur entier différent
19     de 0 sinon
20     """
21     a = 0 # ceci est un commentaire court simple
22     a = a + n
23
24     # Je peux écrire des commentaires
25     # sur plusieurs lignes
26
27     return
```

III. 3. Commentaires en langage OCaml

En OCaml, il n'existe qu'une seule manière d'écrire des commentaires, en les encapsulant entre les symboles `(*` et `*)` :

```

1
2 (** Commentaire long sur plusieurs lignes
3     Utilise par l'outil de documentation automatique ocamlDOC
4
5
6 Fichier exemple pour montrer les commentaires et la doc auto
7 @author G. OLIVIER
8 @version 1.0.0
9 @since 01/09/2022
10 *)
11
12 (**
13 Fonction bete qui ne sert a rien.
14 @param n : un entier qu'on additionnera.
15 @return retourne 0 si tout va bien, ou un code d'erreur entier différé
16    d'un certain nombre de 0 sinon.
17 *)
18
19 let fonction_bete n =
20     let a = ref 0 in (* commentaire court *)
21     a := !a + n      (** Commentaire court pris en compte par ocamlDOC *)
22     (* Ici je peux écrire des commentaires non pris en compte par la doc
23        sur plusieurs lignes
24     .....
25     *)
26
27 ;;
```

III. 4. Génération automatique de documentation

Il est possible de faire en sorte que les commentaires soient utilisés pour générer automatiquement une documentation du code avec une mise en page lisible et agréable (HTML, Latex...). Cela a été fait dans les codes Python, C et OCaml ci-dessus.

En C et en OCaml, il suffit de rajouter un `*` supplémentaire lors de l'écriture du commentaire pour les rendre exploitables respectivement par `doxygen` et `ocamlDOC`. Les mots commençant par `@` seront interprétés pour la mise en page.

En Python, tous les commentaires entre les trois guillemets ouvrants et fermants peuvent être exploités par `doxygen` ou `pydoc3` pour générer automatiquement la documentation.

Montrons, sur l'exemple en langage C, comment générer automatiquement la documentation.

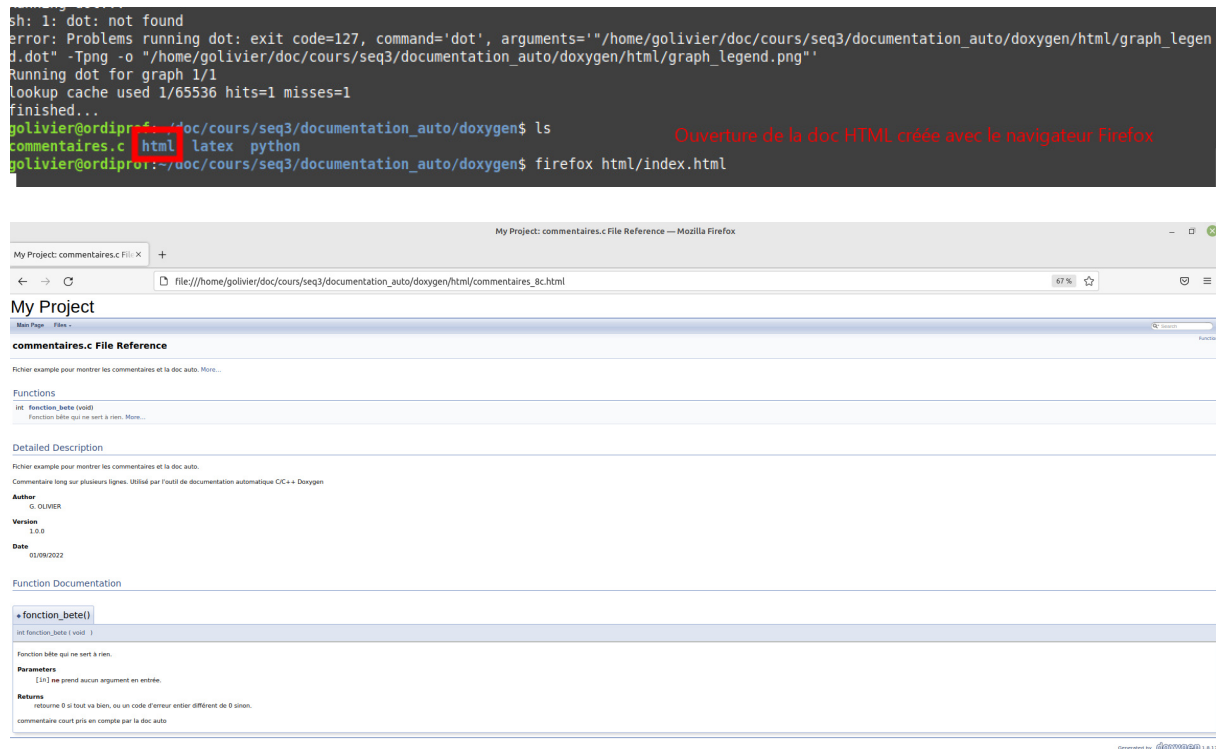
```

golivier@ordipf:~/doc/cours/seq3/documentation_auto/doxygen$ sudo apt-get install doxygen Installation de doxygen si nécessaire (ici non)
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
doxygen est déjà la version la plus récente (1.8.17-0ubuntu2).
0 mis à jour, 0 nouvellement installés, 0 à enlever et 44 non mis à jour.
golivier@ordipf:~/doc/cours/seq3/documentation_auto/doxygen$ ls
commentaires.c python
golivier@ordipf:~/doc/cours/seq3/documentation_auto/doxygen$ doxygen commentaires.c -w html
warning: ignoring unknown tag 'Commentaire' at line 2, file commentaires.c
warning: ignoring unknown tag 'long' at line 2, file commentaires.c
warning: ignoring unknown tag 'sur' at line 2, file commentaires.c
warning: ignoring unknown tag 'plusieurs' at line 2, file commentaires.c
warning: ignoring unknown tag 'lignes' at line 2, file commentaires.c
```

1. Il faut tout d'abord s'assurer que `doxygen` est bien installé et l'installer si ce n'est pas le cas.
2. Il suffit ensuite d'écrire des commentaires dans le fichier code source en C que l'on souhaite documenter. Les commentaires doivent être écrits sous la forme donnée ci-dessus pour pouvoir être correctement interprétés par `doxygen`.

3. Il faut enfin appeler la commande `doxygen` pour lancer l'outil de génération automatique de documentation sur ce code source
4. On peut visualiser la documentation HTML obtenue en ouvrant le fichier racine de la documentation web, appelée `index.html`, avec Firefox. Ces pages web HTML pourront être déposées sur un serveur Web pour être accessibles depuis le monde entier :-)

Voici l'allure de la documentation HTML générée automatiquement par l'outil `doxygen` en lui fournissant en entrée le code source en langage C ci-dessus :



IV. Variables : affectation, type, mutabilité

La programmation impérative est fondamentalement associée aux notions de variable et d'affectation issues des conceptions de Turing et de Von Neumann, et qui sous-tendent l'architecture des machines programmables modernes.

IV. 1. Définition

Définition 2 (Variable)

Une **variable** est une représentation d'une zone de stockage mémoire (qui peut être localisée physiquement en mémoire principale, dans les caches ou les registres du microprocesseur, voir cours séquence 4 à venir). La notion de variable est **centrale** en programmation impérative.

Attention. La notion de variable en informatique n'est pas identique à la notion de variable utilisée en mathématiques.

Une variable est caractérisée par :

son identificateur : il s'agit d'un nom, d'une étiquette qui nous servira à nommer et à faire référence à cette zone mémoire dans la suite du code, cf section II. ;

sa taille en mémoire : il s'agit de la taille de la zone de stockage mémoire représentée par cette variable, généralement donnée en octets ;

son type (= format) : entier, flottant, caractère, adresse mémoire... le type permet de savoir comment interpréter la série de bits qui sera stockée dans la zone mémoire, comme vu dans la première séquence. Bien sûr, le type et la taille mémoire d'une variable sont fortement corrélés, comme nous l'avons vu en séquence 1. Le nom des différents types est relativement standardisé entre les différents langages de programmation ;

son emplacement en mémoire : la zone mémoire concernée, constituée d'un ensemble d'octets voisins, peut être décrite par sa position de départ, appelée **emplacement mémoire**, et par sa **taille** exprimée en octets. Nous verrons plus loin dans le cours que l'emplacement mémoire est décrit à travers une *adresse mémoire*, c'est-à-dire un numéro de bloc dans la mémoire, indépendant de la variable représentée ;

son champ (*scope*) : il s'agit de son champ, de sa zone d'existence dans le code. Une variable est créée, elle vit, est utilisée, puis elle est détruite (automatiquement ou manuellement). Certaines variables persistent pendant toute la durée de l'exécution, d'autres ont une existence très brève à l'échelle d'une petite sous-fonction....

IV. 2. Types simples prédéfinis

Définition 3 (Type)

Le **type** d'une variable correspond au format d'encodage binaire de la donnée qu'elle est destinée à contenir. Les types les plus courants sont les types entier, flottant (simple ou double précision), caractère et adresse mémoire (nous verrons ce que cela signifie).

On rappelle dans la figure ci-dessous le lien entre type et taille de mémoire associée sur les machines actuelles. Les noms utilisés sont ceux du langage C. En Python et OCaml, tous les types entiers sont appelés **int** et tous les types réels sont appelés **float** car l'interpréteur se charge de déterminer et d'allouer l'espace mémoire nécessaire en fonction des besoins et des demandes et d'effectuer des optimisations le cas échéant.

Nom du type en C	Explication	Taille de stockage	Valeurs limites encodables
char	caractère	8 bits	-128 à 127
unsigned char	caractère non signé	8 bits	0 à 255
short int	entier court signé	16 bits	-32768 à 32768
unsigned short int	entier court non signé	16 bits	0 à 65535
int	entier signé	32 bits	-2 147 483 648 à 2 147 483 647
unsigned int	entier non signé	32 bits	0 à 4 294 967 295
float	réel simple précision	32 bits	$\pm 3.4 \times 10^{-38}$ à 3.4×10^{38}
double	réel double précision	64 bits	$\pm 1.7 \times 10^{-308}$ à 1.7×10^{308}
long double	réel long double précision	80 bits	$\pm 3.4 \times 10^{-4932}$ à 3.4×10^{4932}

Attention. En informatique, la virgule dans un nombre à virgule se note toujours avec un point. Prenez l'habitude, au moins en cours d'informatique, de noter la virgule d'un nombre avec un point, comme les anglo-saxons...

IV. 3. Type booléen

Les langages de très haut niveau Python et OCaml définissent un type **booléen**, noté **bool**, qui n'existe pas en C. Ce type peut stocker uniquement deux valeurs correspondant à :

vrai : **True** en Python, **true** en OCaml

faux : **False** en Python, **false** en OCaml

Ce type est utilisé pour définir des variables stockant des résultats de tests (évaluation d'expressions logiques). En C, les types caractères ou entiers sont utilisés pour stocker des valeurs booléennes : la valeur 1 signifiant vrai, la valeur 0 signifiant faux.

IV. 4. Type vide

C'est le type renvoyé par toute commande qui ne renvoie rien, mais qui éventuellement a un effet de bord (modifier la mémoire, produire un affichage...)

Le type vide est un type un peu particulier, car une seule valeur (la valeur « rien » ou « vide » peut avoir ce type.

En OCaml, la valeur vide est notée **()** et le type est noté **unit**. Ce type sert pour les fonctions qui ne retournent rien et créent des effets de bord (modifier la mémoire, produire un affichage...), donc dans les cas où l'on sort du paradigme fonctionnel pur ;

En Python, la valeur vide est notée **None** et le type associé est **NoneType** ;

En C, la valeur est notée **void**, et le type est également noté **void**

IV. 5. Type chaîne de caractère

Définition 4 (Chaîne de caractères)

Une chaîne de caractères est, comme son nom l'indique, une suite de caractères (lettres, symboles de chiffres, caractères blancs...etc). Il s'agit donc de ce que l'on nomme dans le langage courant un **texte**. Le type chaîne de caractère n'est pas un type simple : c'est un **type construit** à partir du type simple caractère. Plus précisément, une chaîne de caractères est un **tableau de caractères**. Chaque case du tableau correspond à 1 octet (8 bits) qui sert à encoder le caractère correspondant.

Un caractère seul est toujours noté avec un simple guillemet '**c**' alors qu'une chaîne de caractères est toujours entourée de double-guillemets '**“ô rage! ô désespoir!”**'.

Il s'agit d'un type à part entière en Python (type **str**) et en OCaml (type **string**).


```
golivier@ordiprof: ~
Fichier Édition Affichage Rechercher Terminal Aide
golivier@ordiprof:~/doc/cours/seq3/documentation_auto/doxygen$ cd
golivier@ordiprof:~$ python3
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> mon_texte="J'aime coder"
>>> type(mon_texte)
<class 'str'>
>>> mot="informatique"
>>> type(mot)
<class 'str'>
>>> print(mot[2])
f
>>>
```

Lancement de la console interactive (REPL) Python

Affichage du caractère stocké dans la case n°2 de la chaîne de caractère stockée dans la variable mot

```
golivier@ordiprof: ~
Fichier Édition Affichage Rechercher Terminal Aide
golivier@ordiprof:~$ ocaml
OCaml version 4.05.0
# let mon_texte="J'aime coder";;
val mon_texte : string = "J'aime coder"
# let mot="informatique";;
val mot : string = "informatique"
# mot.[2];;
- : char = 'f'
#
```

Lancement de la console interactive (ou toplevel ou REPL) OCaml

Affichage du caractère associée à la case n°2 de la chaîne de caractères associée à l'identificateur mot

En C, ce type n'existe pas en tant que tel : une chaîne de caractère est du type `char *` (pointeur vers l'adresse de début d'un tableau de caractères, nous verrons le sens de cette dénomination dans une prochaine séquence)

Exemple 1

Considérons l'encodage mémoire de la chaîne de caractère `'informatique'` : les 8 bits de la case 2 du tableau de cette chaîne de caractère servent à stocker l'encodage ASCII du caractère `'f'` (la numérotation commence toujours à 0 en Python, C et OCaml).

```
golivier@ordiprof:~$ python3
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> mot="informatique"
>>> len(mot)
12
>>> print(mot[2])
f
>>> print(mot[17])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(mot[-3])
q
>>>
```

En Python, -3 pour un indice de tableau signifie 3 en partant de la fin

```

OCaml version 4.05.0

# let mot="informatique";;
val mot : string = "informatique"
# String.length mot;;
- : int = 12
# mot.[2];;
- : char = 'f'
# mot.[17];;
Exception: Invalid_argument "index out of bounds".
# mot.[-3];;
Exception: Invalid_argument "index out of bounds".
#

```

Attention. En Python, le type simple caractère n'apparaît en pratique quasiment jamais car un caractère isolé est considéré comme une chaîne de caractères formée d'un seul caractère. Ce n'est pas le cas en C et en OCaml, qui font la différence entre un caractère seul et une chaîne de caractères.

<pre> golivier@ordiprof:~\$ ocaml OCaml version 4.05.0 # let mon_caractere='g';; val mon_caractere : char = 'g' # </pre>	<pre> golivier@ordiprof:~\$ python3 Python 3.8.10 (default, Jun 22 2022, 20:18: [GCC 9.4.0] on linux Type "help", "copyright", "credits" or "lic >>> mon_caractere='g' >>> type(mon_caractere) <class 'str'> </pre>
---	---

Nous verrons en détail ce qu'est réellement au niveau mémoire une chaîne de caractères dans le chapitre sur les tableaux et les pointeurs.

IV. 6. Affectation d'une valeur à une variable

Une variable est une zone de stockage mémoire et peut être vue comme un contenant avec une étiquette (= un identificateur), dans lequel on va stocker une valeur. Gardez bien en tête cette métaphore du contenant quand il s'agit de variables et d'affectation.

Définition 5 (Opération d'affectation)

Lorsque l'on remplit la zone mémoire associée à la variable, on dit que l'on réalise une opération **d'affectation**, notée avec une flèche vers la gauche \leftarrow en pseudo-code.

Exemple 2

L'opération :

Affecter 5 à la variable a

peut s'écrire de façon plus concise en pseudo-code :

$a \leftarrow 5$

La flèche vers la gauche indique bien que l'on range la valeur 5 **dans** la zone de stockage (variable) nommée **a**.

IV. 6. a. Affectation en langages C et Python.

On voit qu'en Python et en C, l'opération d'affectation est notée avec un signe =.

Attention. La notation = au lieu de \leftarrow peut entraîner de graves confusions car l'opération d'affectation n'a rien à voir avec une assertion d'égalité mathématique.

N'oubliez jamais qu'un signe = dans votre code désigne une opération d'affectation et doit être lu comme \leftarrow : je range la valeur à droite du signe = dans la zone de stockage (variable) à gauche du signe égal.

Exemple 3

`a = toto`

signifie : je copie la valeur contenue dans la zone de stockage (variable) appelée **toto** et je mets cette valeur dans la zone de stockage (variable) appelée **a**.

IV. 6. b. Déclarations en OCaml.

En OCaml, il n'y a pas à proprement parler de variable car la notion de variable vue en tant que contenant est une notion très marquée par le paradigme impératif. Le langage OCaml, avant tout conçu à travers le paradigme de la programmation fonctionnelle, privilégie la notion de valeur immuable à celle de variable : on parle donc de **valeurs** et de déclaration de valeurs (plutôt que d'affectation).

Une valeur est déclarée à l'aide du mot clé **let** et l'identificateur (le nom) de cette valeur est immédiatement indiqué à l'aide du signe = qui, dans ce contexte, a davantage le sens qu'on lui donne habituellement en mathématiques :

`let ma_valeur = -5;;`

Dans le code exemple de l'algorithme d'Euclide, **c** et **d** ne sont pas des valeurs mais des références, notion qui se rapproche davantage de celle d'adresse mémoire d'une variable que nous aborderons en détail dans la séquence suivante.

IV. 7. Typage statique v.s typage dynamique

Définition 6 (Typage statique)

Le type d'une variable peut être fixé une bonne fois pour toute lors de sa création. Dans ce cas, on dit que la variable est typée statiquement. C'est toujours le cas pour toutes les variables en langages C.

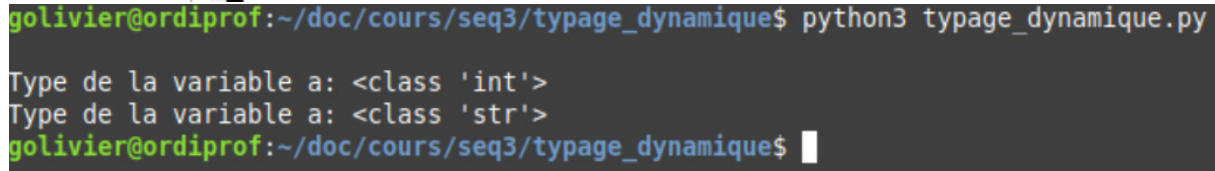
Définition 7 (Typage dynamique)

Le type d'une variable peut être modifié au cours de l'exécution selon le type de valeur qu'elle est destinée à contenir. Dans ce cas, on dit que cette variable est typée dynamiquement. Cela est possible en Python et en OCaml.

Exemple 4

Dans le script Python suivant, la variable `a` change de type au cours de l'exécution :

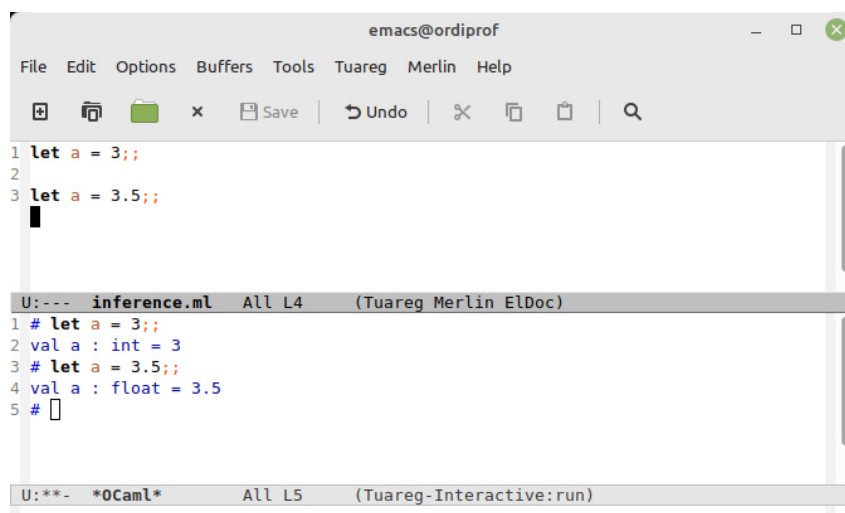
```
1 a=3
2 print("Type de la variable a:", type(a))
3 a='T'
4 print("Type de la variable a:", type(a))
```



```
golivier@ordiprof:~/doc/cours/seq3/typage_dynamique$ python3 typage_dynamique.py
Type de la variable a: <class 'int'>
Type de la variable a: <class 'str'>
golivier@ordiprof:~/doc/cours/seq3/typage_dynamique$
```

Exemple 5

Même chose en OCaml, la variable `a` change de type au cours de l'exécution. Ici, on utilise le module Tuareg d'emacs pour lancer l'interpréteur Ocaml en mode interactif dans une fenêtre Emacs : commande `C-x C-e` lorsque le curseur est devant une ligne de code pour lancer l'interprétation de cette ligne de code.



```
emacs@ordiprof
File Edit Options Buffers Tools Tuareg Merlin Help
[Icons] Save Undo [Icons]
1 let a = 3;;
2
3 let a = 3.5;;
#
```

```
U:--- inference.ml All L4 (Tuareg Merlin ElDoc)
1 # let a = 3;;
2 val a : int = 3
3 # let a = 3.5;;
4 val a : float = 3.5
5 #
```

```
U:**- *OCaml* All L5 (Tuareg-Interactive:run)
```

Définition 8 (Type inféré)

Certains compilateurs et interpréteurs performants sont capables de « deviner » le type des variables en analysant le code source. On dit alors que le type de la variable est inféré par le compilateur ou l'interpréteur. C'est le cas des interpréteurs Python et OCaml.

Exemple 6

Le langage C précise explicitement le type des variables lors de leur déclaration. C'est le cas à la ligne 16 de notre exemple : la variable `c` est déclarée avec le type entier, noté `int`.

A l'inverse, en OCaml et en Python, le type est **inféré** : c'est l'interpréteur qui va deviner le type de ces variables en analysant le code lors de l'interprétation du code.

Attention. Sur de nombreuses sources que l'on trouve sur Internet, il y a confusion entre type dynamique et type inféré. Il s'agit de deux notions différentes, sans lien entre elles.

IV. 8. Variable mutable v.s valeur immuable

Définition 9 (Variable mutable)

Une variable *mutable*, est une variable dont le contenu (la valeur contenue dans la zone mémoire de la variable) peut changer au cours de l'exécution.

Exemple 7

En C et Python, les variables sont mutables, sauf indication contraire : leur contenu peut être modifié à tout moment de l'exécution. C'est le cas dans l'exemple de l'algorithme d'Euclide pour `c`, `d` et `r`.

Sinon, on a affaire à une variable **immuable** ou *immutable* : la valeur contenue dans la variable est affectée une première fois (initialisation) et ne changera (ne variera) plus jamais au cours de l'exécution. En fait, dans ce cas, on ne parlera plus de variable mais de valeur : en effet, il ne s'agit plus vraiment d'une variable puisque la valeur contenue dans la zone de stockage associée à la variable ne changera plus après la première affectation.

Exemples 8

1. En OCaml, les valeurs sont toujours immuables : cela est fondamentalement lié au paradigme de la programmation fonctionnelle qui domine la conception du langage OCaml.
2. En langage C, pour indiquer qu'une variable doit être immuable (et que c'est donc une valeur au sens ci-dessus), on ajoute le mot clé `const` :

```
const int seconds_per_hour = 3600;
```

Exemple 9

Dans le code exemple, nous avons besoin de modifier les valeurs de `c`, `d` et `r` au cours de l'exécution. Dans le code OCaml les identificateurs `c`, `d` et `r` ne désignent donc pas des valeurs mais des **références** (mot-clé `ref`) sur des valeurs, c'est-à-dire, pour simplifier, des adresses mémoires pointant sur des variables qui vont contenir des valeurs modifiables. Pour changer la valeur associée à une référence, on utilise le symbole `:=`. Pour accéder à la valeur pointée par une référence, on utilise le symbole `!` devant le nom de la référence.

V. Programmation structurée

V. 1. Structuration par des fonctions.

Les trois programmes sont **structurés** : ils utilisent la notion de **fonction** pour regrouper des instructions associées à un sous-algorithme. La structuration d'un code par des fonctions se présente en deux étapes.

V. 1. a. Définition de la fonction.

On définit le sous-algorithme sous la forme d'une fonction en indiquant les entrées du sous-algorithme, ses sorties et en détaillant la série d'instructions de l'algorithme. Cette définition est écrite une bonne fois pour toute. Il s'agit d'une définition générale.

Il existe des différences syntaxiques et lexicales, mais l'écriture d'une fonction commence toujours par la définition de l'**entête**² (aussi appelé **prototype**) de la fonction qui indique :

le **nom** donné à la fonction, ici `euclide`

la **liste des arguments (données) d'entrée** requis par la fonction, avec le nom que l'on a choisi pour chacun de ces arguments. Dans le code exemple, il y a deux arguments en entrée ayant pour noms génériques `m` et `n`

Dans notre code exemple, les entêtes des fonctions sont écrites en ligne 14.

En C, on précise également :

le **type de chaque argument d'entrée** : par exemple, dans le code exemple, les entrées `m` et `n` sont des entiers (`int`).

la **type de la valeur retournée** : ici, en C, on précise que la valeur retournée sera un entier, puisque le PGCD de deux nombres est un entier.

Les types des arguments d'entrée et de la valeur de sortie sont **inférés** (= devinés) lors de l'interprétation dans le cas des langages Python et OCaml.

Pour indiquer la valeur à retourner :

En C et en Python, on utilise le mot-clé `return`.

En OCaml, deux points virgules accolés `;;` indiquent la fin de la fonction. En fait, les deux points virgule indiquent la fin de l'évaluation : c'est la dernière valeur évaluée qui est retournée. Dans le code exemple, la dernière valeur évaluée est la valeur associée à `d`. C'est elle qui est la valeur retournée par la fonction `euclide`.

2. entête se dit *header* en anglais : c'est pourquoi les fichiers contenant les déclarations d'entêtes des fonctions ont l'extension `.h` en langage C et C++

Attention. Une fonction retourne toujours soit aucune, soit une unique valeur. Cette valeur peut être de type simple, ou une adresse pointant vers un type construit (tableau, liste, tuple...etc)

En OCaml, l'interpréteur en mode interactif (toplevel) permet de bien voir les types inférés par l'interpréteur et les prototypes des fonctions qui ont été définies.

Exemple 10

Lorsque l'on exécute le code exemple en mode interactif, on voit que l'interpréteur a correctement inféré le type (entier) des arguments d'entrée de la fonction `euclide` et le type entier de la valeur retournée.

```
l1 @return PGCD de m et de n (nombre entier)
l2 **)
l3
l4 let euclide m n =
l5
l6   let c = ref m in
l7   let d = ref n in
l8   let r = ref (!c mod !d) in (** reste de la division **)
l9
l10  while !r <> 0 do
l11
l12    c := !d;
l13    d := !r;
l14    r := (!c mod !d);
l15  done;
l16
l17
l18
l19
l20
l21
l22
l23
l24
l25
l26
l27
l28
l29
l30
l31
l32
l33
l34
l35
l36
l37
l38
l39
l40
l41
l42
l43
l44
l45
l46
l47
l48
l49
l50
l51
l52
l53
l54
l55
l56
l57
l58
l59
l60
l61
l62
l63
l64
l65
l66
l67
l68
l69
l70
l71
l72
l73
l74
l75
l76
l77
l78
l79
l80
l81
l82
l83
l84
l85
l86
l87
l88
l89
l90
l91
l92
l93
l94
l95
l96
l97
l98
l99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
```

Définition 11 (Paramètres formels, paramètres effectifs)

Les **paramètres formels** sont les noms génériques des arguments d'entrée de la fonction (**m** et **n** dans notre exemple) par opposition aux **paramètres effectifs** avec lesquels la fonction est effectivement appelée (56 et 16 dans notre exemple).

Remarque. En C, tous les paramètres formels doivent être associés à des valeurs effectives lors d'un appel de la fonction.

En Python et en OCaml, il est possible d'avoir des arguments d'entrée optionnels. ^a

a. En Python : <https://www.youtube.com/watch?v=6sso3x3sGFs>

En OCaml : <http://pauillac.inria.fr/~remy/poly/mot/21/index.html>

Attention. Lorsqu'une fonction est appelée avec des paramètres effectifs, leur ordre et leur type doivent concorder avec celui des paramètres formels.

Remarque. Les appels de fonction peuvent être **imbriqués** : on peut appeler une fonction dans une autre fonction. On peut même appeler la fonction dans la fonction elle-même (programmation récursive, que nous verrons très prochainement).

V. 2. Point d'entrée du programme

Dans la traduction en langage C du code exemple, on observe la présence d'une fonction **main** qui n'apparaît pas dans les deux autres traductions. Cela est lié à la différence fondamentale entre le C d'une part et Python et OCaml d'autre part : les codes en C sont destinés à être compilés entièrement avant exécution, tandis que les codes en Python et OCaml sont plutôt destinés à être traduits par un interpréteur.

Les codes Python et OCaml sont exécutés pas à pas, à la manière d'un script, alors qu'un code en C est entièrement compilé avant exécution : il faut donc indiquer quel est le **point d'entrée** de l'exécutable qui sera créé à partir du code C.

V. 2. a. Point d'entrée en C : la fonction principale **main**

En langage C, le point d'entrée est toujours la fonction **main**³

Si le programme n'a besoin d'aucune donnée en entrée, on peut écrire la fonction principale sans argument d'entrée, ce qui est le cas dans l'algorithme d'Euclide de l'exemple. Dans ce cas, le prototype du point d'entrée est :

```
int main(void)
```

En C, il est d'usage que la fonction **main** renvoie la valeur 0 si tout s'est bien passé, et un code d'erreur entier (par défaut 1) s'il y a eu un problème.

V. 2. b. Récupération des mots de la ligne de commande en C

La plupart du temps, on a toutefois envie que l'utilisateur puisse appeler plusieurs fois notre programme avec des données différentes et sans changer ou recompiler le code. Par exemple, dans notre code exemple, on aimerait que l'utilisateur puisse changer les 2 nombres d'entrée à chaque appel du programme **euclide** en ligne de commande. Pour cela, on utilise en C la fonction principale avec le prototype suivant :

3. *main* signifie principal, **main** est donc la **fonction principale** du programme.

```
int main(int argc, char ** argv)
```

Cette fonction principale prend en entrée deux éléments :

argc : le nombre de mots de la ligne de commande correspondant au lancement de l'exécutable

argv : la liste de tous les mots de la ligne de commande sous la forme de chaînes de caractères.

Exemple 12

Par exemple, si le programme est exécuté à partir de la ligne de commande :

```
> ./euclide 28 63
```

alors, dans la fonction principale **main** du programme C associé à l'exécutable **euclide**, **argc** contient la valeur 3 car il y a 3 mots dans la ligne de commande :

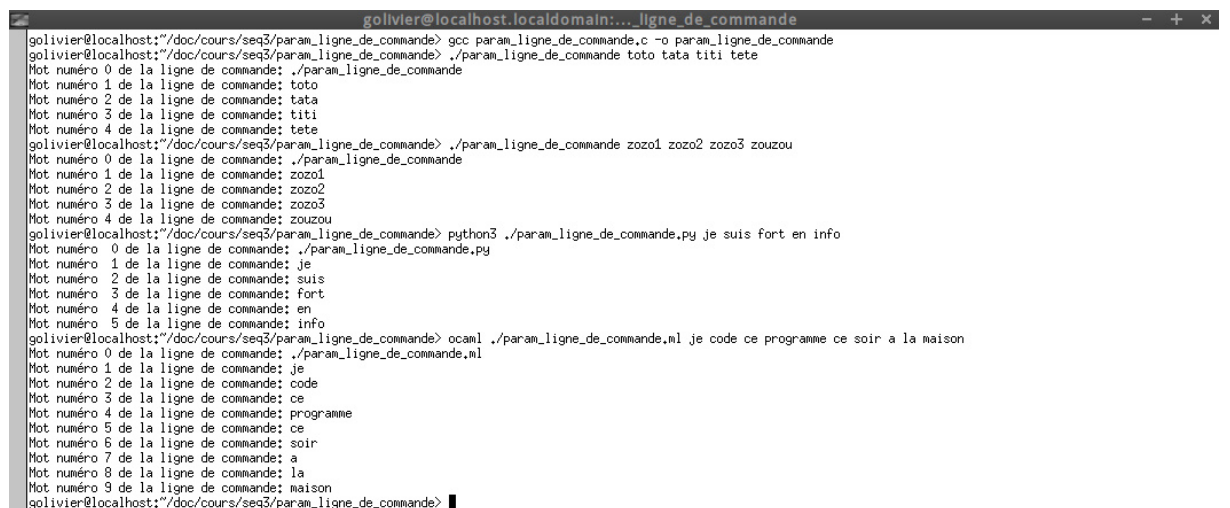
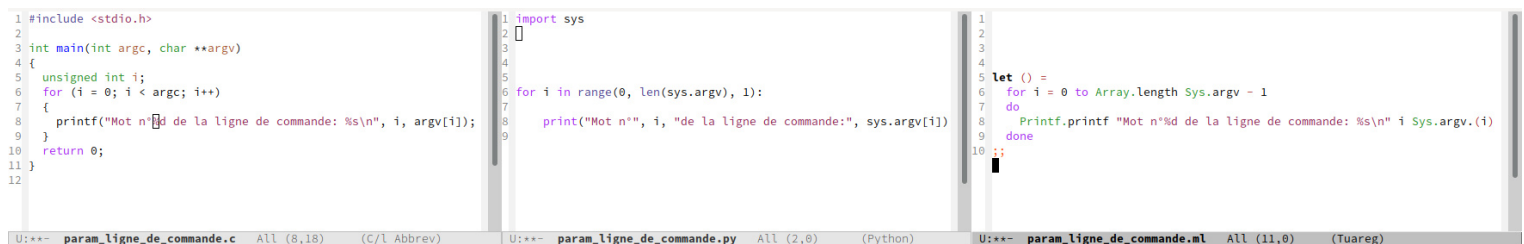
- **argv[0]** contient la chaîne de caractères `./euclide`
- **argv[1]** contient la chaîne de caractères `28`
- **argv[2]** contient la chaîne de caractères `63`

Libre ensuite au programmeur d'utiliser ces mots de la ligne de commande comme valeurs dans son programme.

V. 2. c. Récupération des mots de la ligne de commande en Python et OCaml

Les langages Python et OCaml ont une syntaxe très similaire pour récupérer les mots de la ligne de commande, en faisant appel à une bibliothèque nommée **Sys** en OCaml et **sys** en Python pour *system*.

La figure ci-dessous montre comment récupérer les mots de la ligne de commande dans les 3 langages et montre quelques exemples d'exécution de ces trois codes.



V. 2. d. Fonctions de conversion de types

Attention. Les mots sont récupérés dans le code sous la forme de chaînes de caractères (`char *` en C, `str` en Python et `string` en OCaml). Il faudra utiliser des fonctions de conversion pour les transformer en nombre au niveau de leur stockage en mémoire si les algorithmes qui utilisent ces entrées travaillent sur des nombres.

Exemple 13

Par exemple, si je tape la ligne de commande

```
> ./euclide 28 63
```

`argv[1]` contient la chaîne de caractère ‘‘28’’ et non l’entier 28. Il faut donc explicitement indiquer à la machine qu’elle doit convertir cette chaîne de caractère en un entier. Pour cela, des fonctions de conversion sont mises à disposition.

En C, il faudra aller voir le manuel des fonctions `atoi` et `atof`.⁴

En Python, les fonctions `int()` et `float()` permettent de faire cette conversion.

En OCaml, on ira consulter la documentation des fonctions `int_of_string` et `float_of_string`.

V. 3. Portée (scope) d’un identificateur

Si l’on observe bien les trois codes, tous utilisent le nom de variable `d` dans deux contextes différents :

- à l’intérieur de la fonction `euclide`, pour désigner une variable de travail.
- Et une fois à l’extérieur de cette fonction, dans le programme principal, pour stocker la valeur - le PGCD - retourné par l’appel à la fonction `euclide` (ligne 25).

Comment le compilateur/interpréteur gère-t-il cette homonymie ?

La réponse se trouve dans les mécanismes bas niveau d’appel de fonction que nous verrons en détail dans la séquence suivantes traitant des processus. Mais vous pouvez d’ores et déjà retenir ceci :

Définition 12 (Scope (champ d’action ou portée) d’un objet)

On appelle *scope* ou portée d’un objet informatique la portion de code au sein de laquelle l’association entre l’identificateur de cet objet et l’objet lui même - notamment en mémoire - existe et peut-être utilisée pour l’exécution du programme.

4. En toute rigueur, mieux vaut utiliser les fonctions `strtol` et `strtod` mais leur utilisation requiert des connaissances sur les pointeurs que vous ne possédez pas encore

Définition 13 (Variable locale)

Si la portée d'une variable ou d'une déclaration est limitée à certaines portions de code, on parle de variable ou de déclaration locale.

En dehors de son *scope*, la variable ou la déclaration locale sera détruite :

- la zone mémoire associée ne sera plus accessible grâce à l'identificateur, et pourra même être ré-alloué pour un autre usage complètement différent
- l'identificateur (nom) de la variable ou de la déclaration ne sera plus relié à cette zone mémoire et il pourra même être réattribué pour désigner tout autre chose, éventuellement même une variable d'un type totalement différent ou même un nouvel objet qui n'est même pas une variable ou une déclaration, par exemple une fonction !

Définition 14 (Variable globale)

Une variable ou une déclaration est dite globale si l'association entre l'identificateur et ce qu'il désigne (zone de stockage dans le cas d'une variable, valeur dans le cas d'une déclaration) perdure durant tout la vie du programme.

Attention. Dans tous les langages, toute variable déclarée à l'intérieur d'une fonction est par défaut une variable locale : elle est détruite à chaque fin d'exécution de cette fonction. On parle de variable **locale**.

Exemple 14

Dans le code C ci-dessous, le scope de la variable `mon_age` est limité à la fonction `ma_fonction`. Lors de la compilation (de l'interprétation en OCaml), une erreur est repérée dans la fonction principale car en dehors de la fonction `ma_fonction`, la variable `mon_age` est inconnue.

```

1 #include <stdio.h>
2
3 void ma_fonction()
4 {
5     int mon_age = 38;
6     printf("Mon age dans la fonction: %d\n", mon_age);
7     return;
8 }
9
10 int main(void)
11 {
12     ma_fonction();
13
14     printf("Mon age en dehors de la fonction: %d\n", mon_age);
15
16     return 0;
17 }

```

```

U:--- scope.c All L14 (C/*l Abbrev)
golivier@ordiprof:~/doc/cours/seq3/scope$ gcc scope.c -o scope
scope.c: In function 'main':
scope.c:14:52: error: 'mon_age' undeclared (first use in this function)
14 |     printf("Mon age en dehors de la fonction: %d\n", mon_age);
    |                                                    ^~~~~~
scope.c:14:52: note: each undeclared identifier is reported only once for each function it appears in
golivier@ordiprof:~/doc/cours/seq3/scope$

```

```

1 let ma_fonction () =
2   let mon_age = 18 in
3   Printf.printf "Mon age dans la fonction: %d\n" mon_age;;
4
5 ma_fonction ();;
6
7 Printf.printf "Mon age en dehors de la fonction: %d\n" mon_age;;

```

```

U:--- scope.ml All L8 (Tuareg Merlin ElDoc)
# let ma_fonction () =
#   let mon_age = 18 in
#   Printf.printf "Mon age dans la fonction: %d\n" mon_age;;
# val ma_fonction : unit -> unit = <fun>
# # ma_fonction ();;
# Mon age dans la fonction: 18
# - : unit = ()
# # Printf.printf "Mon age en dehors de la fonction: %d\n" mon_age;;
# Characters 55-62:
#   Printf.printf "Mon age en dehors de la fonction: %d\n" mon_age;;
#                                     ^~~~~~
# Error: Unbound value mon_age
U:**. *OCaml* All L9 (Tuareg-Interactive:run)

```

Exemple 15

Dans le code exemple en C et Python, la variable `d` créée à l'intérieur de la fonction `euclide` est détruite à la fin de cette fonction, c'est une **variable locale**.

L'identificateur `d` est réutilisé en dehors de la fonction `euclide` pour récupérer la valeur retournée par l'appel de fonction.

Remarque. En OCaml, le mot-clé `in` permet d'indiquer que l'expression évaluée sera utilisée dans l'évaluation des expressions suivant le mot-clé `in` et uniquement dans ces expressions : la portée de cette expression est donc limitée. Cela est utilisé dans le code exemple.

V. 4. Structuration des lignes de code

V. 4. a. Séquences

Vision impérative. Pour les langages très influencés par le paradigme de la programmation impérative, il suffit de définir un caractère, appelé séparateur, qui permettra au compilateur ou à l'interpréteur de séparer les instructions lors de la phase d'analyse syntaxique :

En Python, c'est généralement le caractère **retour à la ligne** '`\n`' (généralement invisible par défaut dans l'éditeur de texte) qui permet de séparer les différentes instructions.

En C, ce sont des **points virgules** ; qui servent à marquer la fin de chaque instruction et donc à séparer les différentes instructions.

Remarque. En Python aussi on peut utiliser le point virgule.

Par exemple `x = 1 ; print(x)` est possible sur la même ligne.

Mais pour des question de lisibilité, on réserve toutefois le point virgule aux toutes petites expressions

Remarque. En C, le point virgule à la fin de chaque instruction suffit au compilateur/à l'interpréteur pour séparer les instructions mais les programmeurs aiment souvent revenir à la ligne à la fin de chaque instruction pour plus de lisibilité.

```
1 #include <stdio.h>
2
3 void fonction_peu_lisible(){
4     int a = 3; int b = 5; int c = a*b; printf("Résultat c=%d\n", c); return;
5 }
6
7 int meme_fonction_plus_lisible()
8 {
9     int a = 3;
10    int b = 5;
11    int c = a*b;
12    printf("Résultat c=%d\n", c);
13    return 0;
14 }
15
16 int main(void)
17 {
18     fonction_peu_lisible();
19     meme_fonction_plus_lisible();
20
21     return 0;
22 }
23
24 }
```

U:--- sen instructions.c All 114 (C/*1 Abbrév)

Vision fonctionnelle. Selon le paradigme fonctionnel, l'exécution d'un programme correspond à l'évaluation d'expressions reliées entre elles, comme en mathématiques. Deux expressions d'un programme doivent donc toujours être reliées par un opérateur. Par exemple, lorsque les expressions sont de type `int` (c'est-à-dire que leur évaluation donne un entier), elles peuvent être reliées par `+` ou `*`.

Mais comment faire pour relier deux expressions e_1 et e_2 lorsque e_1 est de type `unit` , c'est-à-dire lorsque e_1 est en fait une instruction au sens de la programmation impérative, créant des effets de bord ?

Dans ce cas, l'opérateur utilisé entre les deux expressions est le point virgule et on parle alors de séquence en lien avec les séquences d'instructions de la programmation impérative. Le point-virgule signifie juste : « effectue l'action décrite par e_1 (avec effet de bord) **et ensuite** évalue l'expression e_2 ».

Attention. L'expression avant le point virgule doit donc toujours être de type `unit` (affichage écran par exemple), sinon une erreur sera relevée lors de l'interprétation.

```
1 let seq_instructions_impur (a) =
2   Printf.printf "a vaut %d\n" a; let b = 3*a in b*4
3 ;;
4
5 seq_instructions_impur (1);;

U:--- sep_instructions.ml All L4 (Tuareg Merlin ElDoc)
1 # let seq_instructions_impur (a) =
2   Printf.printf "a vaut %d\n" a; let b = 3*a in b*4;;
3 val seq_instructions_impur : int -> int = <fun>
4 # seq_instructions_impur (1);;
5 a vaut 1
6 - : int = 12
7 #

U:**- *OCaml* All L7 (Tuareg-Interactive:run)
```

Remarque. L'utilisation du point-virgule pour séparer des instructions dans une séquence en OCaml est une concession accordée au programmeur pour faire de la programmation impérative dans un langage essentiellement influencé par le paradigme fonctionnel. Elle doit être utilisée de façon exceptionnelle, notamment pour demander des affichages écran. En général, quand on ne sait pas comment relier l'évaluation de deux expressions, c'est le mot-clé `in` qui permet de faire ce que l'on souhaite.

Attention. En OCaml, il y a une subtilité : la séquence (le point-virgule) est prioritaire sur le `in`. De même que $2 + 3 * 2$ est automatiquement compris comme $2 + (3 * 2)$.

Par exemple, la formule `let x = 2 in ma_fonction x ; x+1` est automatiquement comprise comme : `let x = 2 in (ma_fonction x ; x+1)`.

V. 4. b. Blocs

Définition 15 (Blocs d'instructions ou d'évaluations)

A l'intérieur d'une fonction, les instructions/évaluations sont regroupées par **blocs** : bloc d'instructions appartenant à une boucle en programmation impérative, bloc d'instructions ou d'évaluations effectuées sous condition...

Ces blocs sont indiqués de différentes manières en fonction des langages :

En C : par des accolades encapsulant le bloc

En Python : par un décalage horizontal en début de ligne, appelé **indentation** et compté en caractères `'\t'` de tabulations (touche Tab), pour chaque ligne/instruction

En OCaml : par des parenthèses ou par les mots-clés `begin` et `end` encadrant les évaluations appartenant à un bloc.

```

1 let calcul_prix_promo kg_fraises kg_pommes =
2   if kg_pommes >= 3.0 then
3     begin
4       let prix_pommes = kg_pommes *. 2.10 in
5       let prix_fraises = kg_fraises *. 7.0 in
6       prix_pommes +. prix_fraises
7     end
8   else
9     begin
10      let prix_pommes = kg_pommes *. 2.90 in
11      let prix_fraises = kg_fraises *. 7.0 in
12      prix_pommes +. prix_fraises
13    end
14 ;;
15
16 calcul_prix_promo 1.0 0.0;;

```

U:--- blocs.ml All L15 (Tuareg Merlin ElDoc)

```

1 # let calcul_prix_promo kg_fraises kg_pommes =
2   if kg_pommes >= 3.0 then
3     begin
4       let prix_pommes = kg_pommes *. 2.10 in
5       let prix_fraises = kg_fraises *. 7.0 in
6       prix_pommes +. prix_fraises
7     end
8   else
9     begin
10      let prix_pommes = kg_pommes *. 2.90 in
11      let prix_fraises = kg_fraises *. 7.0 in
12      prix_pommes +. prix_fraises
13    end;;
14 val calcul_prix_promo : float -> float -> float = <fun>
15 # calcul_prix_promo 1.0 0.0;;
16 - : float = 7.
17 #

```

U:**. *OCaml* All L17 (Tuareg-Interactive:run)

```

1 let calcul_prix_promo kg_fraises kg_pommes =
2   if kg_pommes >= 3.0 then
3     (
4       let prix_pommes = kg_pommes *. 2.10 in
5       let prix_fraises = kg_fraises *. 7.0 in
6       prix_pommes +. prix_fraises
7     )
8   else
9     (
10      let prix_pommes = kg_pommes *. 2.90 in
11      let prix_fraises = kg_fraises *. 7.0 in
12      prix_pommes +. prix_fraises
13    )
14 ;;
15
16 calcul_prix_promo 1.0 0.0;;

```

-.:**. blocs_parentheses.ml All L11 (Tuareg Merlin ElDoc)

```

1 # let calcul_prix_promo kg_fraises kg_pommes =
2   if kg_pommes >= 3.0 then
3     (
4       let prix_pommes = kg_pommes *. 2.10 in
5       let prix_fraises = kg_fraises *. 7.0 in
6       prix_pommes +. prix_fraises
7     )
8   else
9     (
10      let prix_pommes = kg_pommes *. 2.90 in
11      let prix_fraises = kg_fraises *. 7.0 in
12      prix_pommes +. prix_fraises
13    );;
14 val calcul_prix_promo : float -> float -> float = <fun>
15 # calcul_prix_promo 1.0 0.0;;
16 - : float = 7.
17 #

```

U:**. *OCaml* Bot L33 (Tuareg-Interactive:run)

float

Remarque. En Python, l'indentation des blocs est obligatoire.

En C, les accolades suffisent au compilateur pour associer les instructions dans un bloc, mais les programmeurs aiment également rajouter ces indentations pour améliorer la lisibilité du code.

En OCaml, les mots clés et/ou les parenthèses suffisent à l'interpréteur pour repérer l'ordre et l'enchaînement des évaluations, mais les programmeurs aiment également rajouter ces indentations pour améliorer la lisibilité du code.

V. 5. Une fonction particulière : fonction d'affichage de texte dans la console.

Attention. En tout premier lieu, il est important de bien **faire la différence** entre :

- d'une part, le fait pour une fonction de **retourner une valeur**, qui pourra donc être utilisée par d'autres instructions du programme
- d'autre part, le fait d'**afficher une valeur à l'écran**.

En ligne 34 des codes d'exemple, une instruction d'affichage à l'écran dans le console est utilisé.

En langage C, il s'agit de la fonction `printf`. Pour obtenir des informations sur cette fonction, il est possible d'appeler le manuel en ligne de commande `man` en sélectionnant la page `3p` (3 pour indiquer qu'il s'agit d'une fonction présente dans une librairie système et `p` pour POSIX), comme indiqué sur la figure ci-dessous :

```
golivier@localhost:~/doc/cours> man printf
Man: find all matching manual pages (set MAN_POSIXLY_CORRECT to avoid this)
* printf (1)
  printf (3)
  printf (1p)
  printf (3p)
  Printf (3o)
Man: What manual page do you want?
Man: 3p
-
```

Le manuel s'affiche à l'écran :

```
PRINTF(3P)                                POSIX Programmer's Manual                                PRINTF(3P)

PROLOG
This manual page is part of the POSIX Programmer's Manual. The Linux implementation
of this interface may differ (consult the corresponding Linux manual page for details
of Linux behavior), or the interface may not be implemented on Linux.

NAME
printf - print formatted output

SYNOPSIS
#include <stdio.h>

int printf(const char *restrict format, ...);

DESCRIPTION
Refer to fprintf().
```

La fonction `printf` se trouve donc dans la bibliothèque système et la description de son entête, nécessaire à la génération des fichiers objets, se trouve dans le fichier `stdio.h`⁵. Cela explique la nécessité de la ligne 1 dans le code exemple en C.

5. Les fichiers d'entête, comme `stdio.h`, sont stockés dans le répertoire système `/usr/include` de votre système d'exploitation Linux, vous pouvez aller voir et ouvrir ce fichier avec Emacs

```
#include <stdio.h>
```

Cette ligne commence par un `#` car il s'agit d'une directive destinée à l'algorithme de pré-traitement (pré-processing) appliqué avant la compilation et l'assemblage dans la chaîne de compilation (cf Séquence 2). Cette directive de pré-processing indique au compilateur dans quel fichier il pourra trouver les informations relatives à la fonction `printf`.

La fonction `printf` permet d'afficher du texte à l'écran en formatant les valeurs à afficher. Voici le prototype de cette fonction :

```
int printf(const char *format, ...);
```

Les pointillés signifient que c'est une **fonction variadique**, qui peut prendre un nombre variable de paramètres.

L'entrée de la fonction, appelée `format` dans le prototype, représente, comme son nom l'indique, un texte formaté. Son principe est le suivant : à chaque fois qu'il y a un `%`, `printf` regarde la lettre qui suit ce `%` et écrit la variable qui correspond dans les paramètres. Si c'est le *i*ème `%`, `printf` regarde le (*i*ème + 1) paramètre passé en argument de la fonction `printf`.

Dans ce cas, la lettre qui suit les « `%` » dans le texte format correspond au type (= format) qui doit être utilisé pour interpréter et afficher correctement la valeur associée :

Interprétation à utiliser pour l'affichage	Lettre à mettre après le %
entier	%d
réel simple précision	%f
réel double précision	%lf
caractère ASCII	%c
entier hexadécimal	%x
entier octal	%o
entier non signé	%u

Voici quelques exemples :

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = -3;
6     int b = 51;
7     float ma_valeur = 2.3471;
8     char caract = 'b';
9
10    printf("a vaut %d et b vaut %d\n", a, b);
11    printf("La multiplication de a et b vaut %d\n", a*b);
12    printf("b affiché en hexadécimal: %x\n", b);
13    printf("ma_valeur (réel) vaut: %f\n", ma_valeur);
14    printf("caract affiché normalement %c\n", caract);
15    printf("caract affiché en octal %o et en hexadécimal %x\n", caract, caract);
16
17    return 0;
18 }
```

```
golivier@localhost:~/doc/cours/seq3> gcc printfex.c -o printfex
golivier@localhost:~/doc/cours/seq3> ./printfex
a vaut -3 et b vaut 51
La multiplication de a et b vaut -153
b affiché en hexadécimal: 33
ma_valeur (réel) vaut: 2.347100
caract affiché normalement b
caract affiché en octal 142 et en hexadécimal 62
golivier@localhost:~/doc/cours/seq3> █
```

Pour les deux autres langages du programme :

En OCaml, la fonction d’affichage `printf` est contenue dans le module (bibliothèque) `Printf`. La syntaxe est la même qu’en C. C’est une fonction qui ne renvoie rien (valeur retour de type `unit` et qui crée des effets de bords. Elle est donc impure dans le cadre de la programmation fonctionnelle.

En Python, la fonction `print` gère automatiquement les formats d’affichage.

Les codes exemple montrent comment les utiliser en OCaml et Python.

VI. Opérateurs

Cette partie ne présente que quelques opérateurs utiles, sans prétendre à l’exhaustivité. Nous avons déjà vu un opérateur, l’opérateur d’affectation qui s’écrit en C et Python avec un signe `=` (dangereux !)

VI. 1. Opérateurs arithmétiques

Ce sont les **opérateurs binaires** usuels de l’**addition** `+`, de la **soustraction** `-`, de la **multiplication** `*`, de la **division** `/`.

L’**opérateur unaire** de **changement de signe** est `-`. Selon sa position, le système sait reconnaître la nature de `-`.

Opération	Python	C	OCaml
Addition de deux entiers	<code>+</code>	<code>+</code>	<code>+</code>
Addition de deux réels	<code>+</code>	<code>+</code>	<code>+. </code>
Soustraction de deux entiers	<code>-</code>	<code>-</code>	<code>-</code>
Soustraction de deux réels	<code>-</code>	<code>-</code>	<code>-. </code>
Multiplication de deux entiers	<code>*</code>	<code>*</code>	<code>*</code>
Multiplication de deux réels	<code>*</code>	<code>*</code>	<code>*. </code>
Division de deux entiers	<code>//</code>	<code>/</code>	<code>/</code>
Division de deux réels	<code>/</code>	<code>/</code>	<code>/. </code>
Reste de la division euclidienne de deux entiers	<code>%</code>	<code>%</code>	<code>mod</code>
Incrémenter d’un nombre entier	<code>++</code>	<code>++</code>	non prévu
Décrémenter d’un nombre entier	<code>--</code>	<code>--</code>	non prévu

Le langage OCaml fait donc la différence entre des opérations entre entiers et des opérations entre nombres réels.

En langage C et Python, les opérateurs arithmétiques sont notés de façon identique qu’il s’agisse d’opérations entre entiers ou entre flottants (sauf pour la division euclidienne en Python).

Attention. Attention toutefois car cela peut être source de confusion, notamment pour la division, qui peut donner des résultats très différents selon que ses opérandes sont considérées comme des nombres réels ou entier (division entière) :

```
int x;
x = 10 / 4;
la valeur 2 est affectée à x alors que dans le code :
float x;
x = 10 / 4;
la valeur 2.5 est affectée à x.
```


VI. 2. Opérateurs de comparaison.

Ce sont les opérateurs permettant de comparer des expressions, notamment pour l'écriture de tests. La valeur renvoyée est de type `bool` en Python et OCaml, et de type `int` voire `char` (pour optimiser l'espace mémoire en stockant sur 1 octet au lieu de 4) en langage C. Nous détaillons ci-dessous les opérateurs de comparaison entre des nombres (types entiers ou flottants) :

Comparaison	Python	C	OCaml
strictement supérieur	<code>></code>	<code>></code>	<code>></code>
supérieur ou égal	<code>>=</code>	<code>>=</code>	<code>>=</code>
strictement inférieur	<code><</code>	<code><</code>	<code><</code>
inférieur ou égal	<code><=</code>	<code><=</code>	<code><=</code>
égal (au sens mathématique)	<code>==</code>	<code>==</code>	<code>=</code> (très dangereux!)
différent (au sens mathématique)	<code>!=</code>	<code>!=</code>	<code><></code>

Un test est effectué à la ligne 20 de notre exemple sur Euclide.

Remarque. Nous verrons qu'il est également possible de faire des tests de comparaison au sens informatique : dans ce cas, le test ne porte pas sur les valeurs contenues dans les variables mais sur les variables elles-mêmes : deux variables seront égales si elles désignent le même emplacement mémoire, si elles ont la même adresse.

VI. 3. Opérateur logiques.

Ils permettent de combiner des expressions logiques, en particulier pour les tests des structures de contrôle. Les opérateurs qui nous seront utiles cette année sont :

Comparaison	Python	C	OCaml
ET logique	<code>and</code>	<code>&&</code>	<code>&&</code>
OU logique	<code>or</code>	<code> </code>	<code> </code> ou bien <code>or</code>
NON logique	<code>not</code>	<code>!</code>	<code>not</code>

VII. Structures de contrôle

La programmation impérative est fondamentalement associée à la notion d'exécution séquentielle propre aux architectures de Von Neumann. Dans ce modèle, un algorithme est une série d'instructions (d'ordres) exécutées les unes à la suite des autres. Il en découle certains **invariants algorithmiques**, aussi appelés **structures de contrôle** propres au paradigme de la programmation impérative. Au niveau machine, ces structures de contrôles sont liées à la notion de **branchement**, aussi appelé **saut** (*jump*, *goto*) qui permet de modifier le compteur ordinal pour poursuivre l'exécution en revenant à un autre endroit en amont dans le code.

VII. 1. Branchements conditionnels

Les branchements conditionnels permettent de modifier le flux d'exécution séquentiel d'un code si une condition est vérifiée.

```
1 Si test alors
2   | Bloc d'instructions exécutées si le test est vérifié
3 Sinon
4   | Bloc d'instructions exécutées si le test n'est pas vérifié
```

La Figure 5 ci-dessous montre la syntaxe des instructions conditionnelles dans les 3 langages.

VII. 2. Boucles

Les boucles permettent de modifier le flux d'exécution séquentiel d'un code en créant des boucles qui permettent de répéter un bloc d'instructions jusqu'à ce qu'une condition soit vérifiée. Une boucle **tant que** apparaît dans notre exemple en ligne 20. Les syntaxes sont très similaires dans les trois langages.

Notre exemple montre la syntaxe des boucles de type *tant que* dans les 3 langages.

La Figure 6 ci-dessous montre la syntaxe des boucles de type *pour... allant de....* dans les 3 langages.

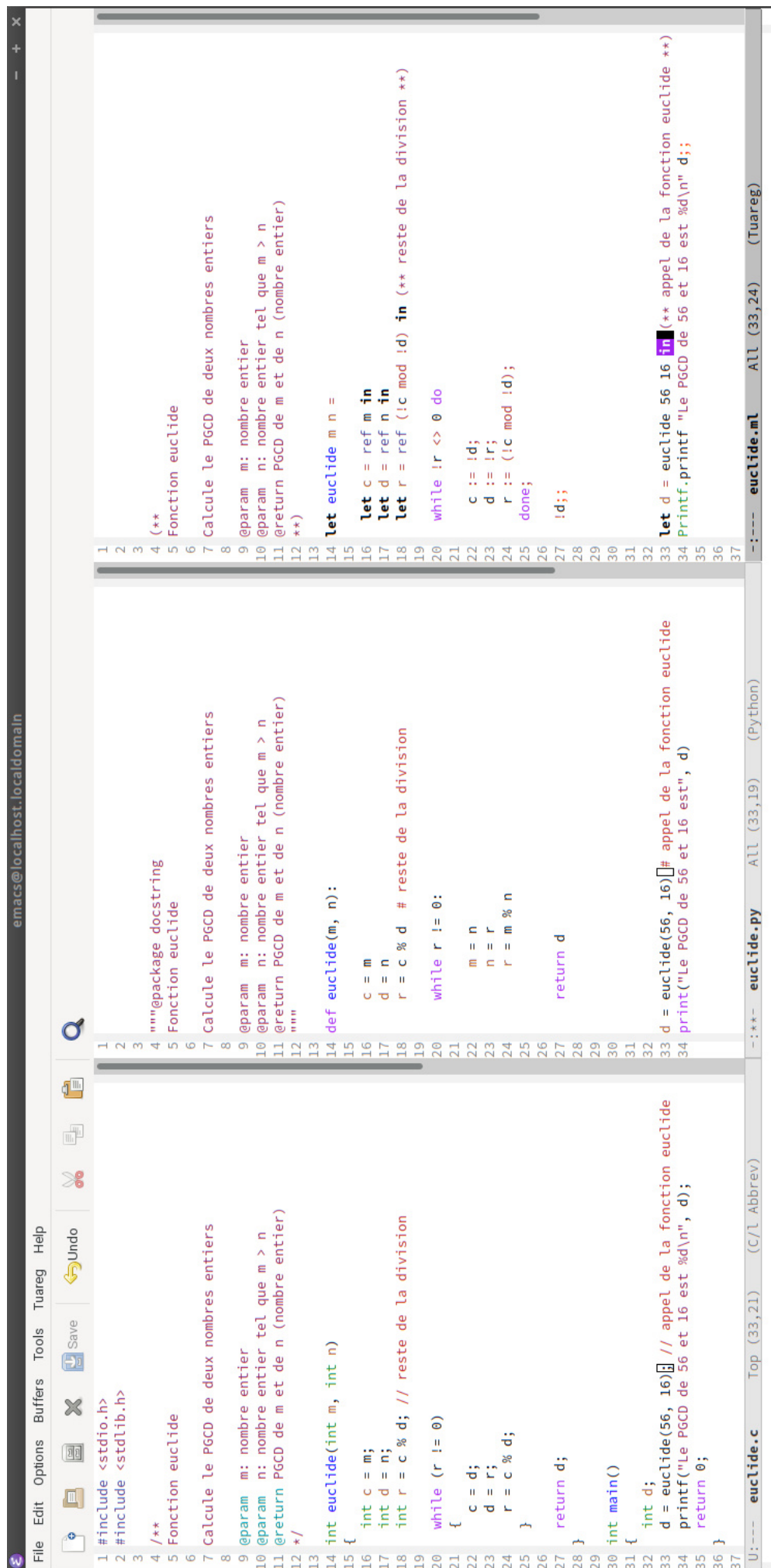


FIGURE 1 – De gauche à droite : écriture de l’algorithme d’Euclide en langage C, Python et OCaml sous emacs.

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

FIGURE 2 – Les 32 mots-clé du langage C dans la norme ANSI

and	as	assert	break	class	continue	def	del
elif	else	except	exec	finally	for	from	global
if	import	in	is	lambda	not	or	pass
print	raise	return	try	while	with	yield	

FIGURE 3 – Les 32 mots-clé du langage Python

and	begin	do	done	downto	else	end	false
for	if	in	let	match	mutable	of	open
ref	then	to	try	true	type	while	with...

FIGURE 4 – Quelques mots clés du langage OCaml

<pre> 1 #include <stdio.h> 2 3 int main(void) 4 { 5 int a = 1; 6 if (a > 3) 7 { 8 printf("a est plus grand que 3\n"); 9 } 10 else if (a > 2) // cas alternatif facultatif 11 { 12 printf("a est plus grand que 2\n"); 13 } 14 else 15 { 16 printf("a est plus petit que 2\n"); 17 } 18 return 0; 19 } 20 </pre>	<pre> 1 2 3 4 5 let a = 1 in 6 if (a > 3) then 7 8 Printf.printf "a est plus grand que 3\n" 9 10 else if (a > 2) then 11 12 Printf.printf "a est plus grand que 2\n" 13 14 else 15 16 Printf.printf "a est plus petit que 2\n" 17 18 19 ;; </pre>	<pre> 1 2 3 4 5 let a = 1 in 6 if (a > 3) then 7 8 Printf.printf "a est plus grand que 3\n" 9 10 else if (a > 2) then 11 12 Printf.printf "a est plus grand que 2\n" 13 14 else 15 16 Printf.printf "a est plus petit que 2\n" 17 18 19 ;; </pre>
<p>Language C</p>	<p>Language Python</p>	<p>Language OCaml</p>

FIGURE 5 – Branchements conditionnels dans les trois langages C, Python et OCaml.

<pre> 1 #include <stdio.h> 2 3 int main(void) 4 { 5 unsigned int i; 6 float sum = 1.5; 7 8 for (i = 0; i < 10; i++) 9 { 10 sum = sum + 2.5; 11 printf("Iteration %d : sum contient %f\n", i, sum); 12 } 13 14 return 0; 15 } </pre>	<p>Language C</p>
<pre> 1 2 3 4 5 6 sum = 1.5 7 8 for i in range(0, 10, 1): 9 10 sum = sum + 2.5 11 print("Iteration ", i, ": sum contient", sum); </pre>	<p>Language Python</p>
<pre> 1 2 3 4 5 6 let sum = ref 1.5 in 7 8 for i = 0 to 10 9 do 10 sum := !sum +. 2.5; 11 Printf.printf "Iteration %d : sum contient %f\n" i !sum; 12 done 13 14 ;; 15 </pre>	<p>Language OCaml</p>

FIGURE 6 – Boucle for dans les trois langages C, Python et OCaml.