# TP4

Langage C (LC4)

semaine du 22 février

## 1 Premiers pas avec make

### 1.1 L'outil make sans Makefile

Considérez un ou plusieurs programmes de vos TPs précédents. Verifiez le sens de l'option -c de gcc, puis

- observez le temps d'exécution de la commande gcc -c <votre programme>.c (commande time),
- puis observez le temps d'execution de la commande make <votre programme>.o.
- Effacez le fichier <votre programme>.o de votre répertoire courant et recommencez l'opération ci-dessus.

**Question 1.** Qu'en concluez vous à propos de l'utilisation de make ?

### 1.2 Premier Makefile

Considérez les trois programmes suivants, et copiez les dans trois fichiers distincts que vous nommerez respectivement : hello.c, hello.h, main.c.

```c
#include <stdio.h>

void hello()
{
  printf("Hello World\n");
}
```

```c
#ifndef H_GL_HELLO
#define H_GL_HELLO

void hello();

#endif
```

```c
#include <stdio.h>
#include "hello.h"

int main()
{
  hello();
  return 0;
}
```

**Question 2.** Une fois compilé que produit l'execution de ce programme ?

**Question 3.** À quoi semble servir la suite de directives **`#ifndef`**, **`#define`**, et **`#endif`** ?

**Question 4.** Après avoir consulté un manuel ou un tutoriel sur `make`, (le plus complet est sans doute `http://www.gnu.org/software/make/manual/make.html`), mais il y en a de beaucoup plus simples comme `http://www.april.org/files/groupes/doc/make/make.html`, établissez un ordre de dépendance entre les fichiers ci-dessus et les fichiers intermédiaires que vous pourriez construire au cours de la compilation (par exemple les fichiers objets).

**Question 5.** Vous êtes désormais en mesure d'écrire un `Makefile` simple pour compiler le programme, ci-dessus, tout en générant tous les fichiers objets intermédiaires.

**Question 6.** Améliorez votre `Makefile` afin qu'il vous permette aussi de nettoyer votre répertoire, c'est à dire d'en retirer tous les fichiers objets. Il s'agit d'écrire une règle supplémentaire dont la cible sera conventionnellement appelée `clean`.

**Question 7.** Ajoutez une autre règle qui permette de retirer tous les fichiers générés au cours de la compilation. Cela vous permettra de reconstruire complètement vos programmes.

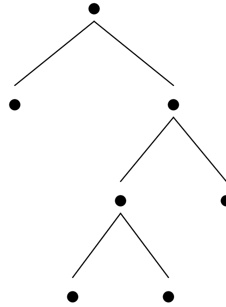
Il est ainsi recommandé de créer un Dossier spécifique et un `Makefile` spécifique à ce dossier. Pour ne pas aller trop vite nous vous fournissons un schéma de Makefile que vous pouvez modifier pour compiler vos exercices.

```
OBJ= mesfichiers.o

exo: $(OBJ)
	gcc -o exo $^

.c.o:
	gcc -o $@  -c $<

clean:
	rm -rf *.o

mrproper: clean
	rm -rf exo
```

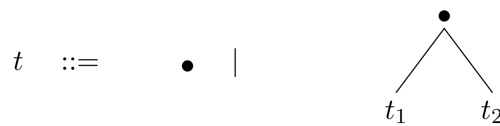
**Question 8.** Avant de passer à la suite essayez de comprendre le sens du `Makefile` proposé.

## 2 Arbres binaires

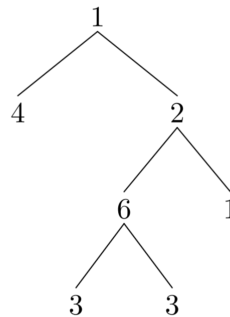
Un arbre binaire ressemble à cela :



De façon générale un arbre binaire $t$ c'est soit une feuille, soit un nœud auquel sont rattachés deux sous arbres $t_1$ et $t_2$ :




On peut aussi étiqueter les nœuds avec des entiers :




Pour représenter de tels objets en $C$ on va utiliser la structure de donnée suivante :

```
struct noeud {
  int val;            // etiquette
  struct noeud *g;    // sous arbre gauche
  struct noeud *d;    // sous arbre droit
};

typedef struct noeud *arbre;
```

Un nœud possède une étiquette `val` de type **`int`** et deux pointeurs vers ses sous arbres (gauche et droit). Les pointeurs auront tous les deux la valeur `NULL` lorsque le nœud est une feuille.

**Question 9.** Écrivez une fonction `arbre feuille(int val)` qui crée un arbre constitué d'une feuille étiquetée par la valeur `val`.

**Question 10.** Écrivez une fonction `arbre combine(int val, arbre t_g, arbre t_d)` qui crée l'arbre constitué d'un nœud étiqueté par `val` qui admet pour sous arbres `t_g` et `t_d`.

**Question 11.** Écrivez une fonction `void free_arbre(arbre t)` qui libère la mémoire allouée pour l'arbre `t`.

**Question 12.** Écrivez une fonction `int somme(arbre t)` qui renvoit la somme des étiquettes d'un arbre `t`.

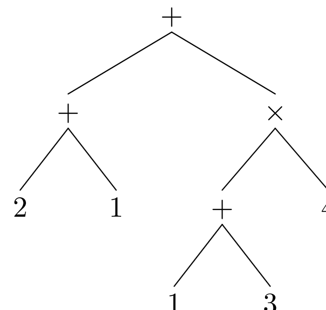
**Question 13.** Écrivez une fonction `void print_arbre(arbre t, int indent)` qui affiche l'arbre `t`. L'arbre donné en exemple sera affiché de la façon suivante (pour `indent` = 0) :

```
1
 4
 2
  6
   3
   3
  1
```

Le paramètre `indent` sert lors des appels récursifs à rajouter le nombre correspondant d'espaces en tête de ligne.

## 3 Mini-calculatrice (s'il vous reste du temps...)

Pour représenter des expressions arithmétiques, on considère des arbres binaires dont les feuilles sont étiquetés par des entiers et dont les nœuds sont étiquetés par $+$ ou $\times$ :




**Question 14.** Définissez une structure de donnée adaptée à ce type d'objet, ainsi qu'un type `expr` pour les désigner.

**Question 15.** Écrivez le constructeur `expr feuille(int val)`, le constructeur `expr combine(int operateur, expr t_g, expr t_d)`, et le destructeur `void free_expr(expr t)` correspondants.

**Question 16.** Écrivez une fonction `int eval(expr t)` qui évalue l'expression arithmétique représentée par l'arbre `t`. Dans l'exemple cette expression est $(2+1)+((1+3)\times 4)$, et la fonction doit renvoyer 19.

**Question 17.** Écrivez une fonction `arbre parse(char **s)` qui construit un arbre à partir d'une chaîne de caractères `*s` de la forme :

```
((2+1)+((1+3)*4))
```

dans laquelle les parenthèses sont obligatoires. Cette fonction renverra la valeur `NULL` si la chaîne contient des caractères incorrects, ou si elle est mal parenthésée. De plus, elle positionnera le pointeur `*s` après le dernier caractère lu.