Langage C – TD2 – Expressions et opérateurs

Julien SEINTURIER / Université de Toulon (julien.seinturier@univ-tln.fr / http://www.seinturier.fr)

1. Expressions arithmétiques

Exercice 1.1.
Citer les cinq opérateurs arithmétiques du langage C. Préciser leur sémantique et leur nature (unaire, binaire, …).

Les cinq opérateurs arithmétiques du langage C sont :

Opérateur	Sémantique	Nature
+	Addition de deux nombres	Binaire
-	Soustraction de deux nombres	Binaire
*	Multiplication de deux nombres	Binaire
/	Division de deux nombres	Binaire
%	Modulo de deux nombres (reste de division)	Binaire

Exercice 1.2.
Ecrire de 4 façons différentes une expression qui affecte à une variable x sa valeur plus 1 (incrémentation de 1).

Voici 4 façons d'incrémenter de 1 la valeur d'une variable x.

`x = x + 1`

`x += 1`

`x++`

`++x`

Exercice 1.3.
Soit une variable x de valeur 1. Que retournent les expressions `x--`, `--x` et `x -= 1` ? Quelle sera la valeur de x après chaque expression (les expressions sont évaluées indépendamment) ?

`x--` retourne 1

`--x` retourne 0

`x -= 1` retourne 0

Les opérateurs unaires préfixés ++x et `--x` retournent la valeur de x après modification. Les opérateurs unaires postfixés x++ et `x--` retournent la valeur de x avant modification.

Dans chaque cas, x vaut 0 après chacune des expressions.

Un programme pour vérifier :

```
#include <stdio.h>

int main()
{

    int x = 1;

    printf("x avant : %d\n", x);
    printf("L'expression x = x - 1 retourne %d\n", x= x - 1);
    printf("x apres : %d\n", x);
    printf("\n");

    x = 1;
    printf("x avant : %d\n", x);
    printf("L'expression x -= 1 retourne %d\n", x-= 1);
    printf("x apres : %d\n", x);
    printf("\n");

    x = 1;
    printf("x avant : %d\n", x);
    printf("L'expression x-- retourne %d\n", x--);
    printf("x apres : %d\n", x);
    printf("\n");

    x = 1;
    printf("x avant : %d\n", x);
    printf("L'expression --x retourne %d\n", --x);
    printf("x apres : %d\n", x);

    return 0;
}
```

Exercice 1.4.
Soit un opérateur arithmétique o (pouvant être `+`, `-`, `*`, `/` ou `%`) et soit les types `char`, `short`, `int`, `float` et `double`. De quel type est l'opérateur arithmétique o pour chaque couple de type possible ? En déduire une règle de conversion de type pour les opérateurs arithmétiques.

O	char	short	int	float	double
char	int	int	int	float	double
short	int	int	int	float	double
int	int	int	int	float	float
float	float	float	float	float	double
double	double	double	double	double	double

Un opérateur arithmétique retourne toujours le type le plus petit pouvant contenir son résultat, sauf pour `char` et `short`. Ces deux types étant changés en `int` avant l'évaluation de l'opérateur.

2. Expressions logiques

Exercice 2.1.
Quel sont les équivalents numériques des valeurs booléennes VRAI et du FAUX en langage C ?

En langage C, toute valeur numérique différente de 0 est équivalente au VRAI. Toute valeur numérique égale à 0 est équivalente au FAUX.

Exercice 2.2.
Citer les opérateurs logiques du langage C et expliciter leur sémantique.

Le langage C possède 3 opérateurs logiques :
- ! qui représente la négation logique (non)
- && qui représente la conjonction logique (et)
- || qui représente la disjonction logique (ou)

Exercice 2.3.
Donner les tables de valeur des opérateurs logiques du langage C.

Table du non :
!VRAI = 0
!FAUX = 1

Table du et :

&&	VRAI	FAUX
VRAI	1	0
FAUX	0	0

Table du ou :

||	VRAI	FAUX
VRAI	1	1
FAUX	1	0

Exercice 2.4.
Soit a et b deux variables. Ecrire en langage C l'expression logique *non (a et b) ou non b et non a*. Quel problème peut se poser lors de l'évaluation de l'expression ?

```
!(a && b) || !b && !a
```

Le problème pouvant se poser est l'ordre des opérateurs à évaluer. Plusieurs interprétations sont possibles :

```
!(a && b) || (!b && !a)

(!(a && b) || !b) && !a

!(a && b) || !(b && !a)
```

…

3. Expressions relationnelles

Exercice 3.1.
Citer les six opérateurs relationnels du langage C et expliciter leur sémantique et leurs valeurs de retour.

Opérateur	Sémantique	Valeur de retour
`a == b`	Retourne VRAI si la valeur de a est égale à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX
`a > b`	Retourne VRAI si la valeur de a est strictement supérieure à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX
`a < b`	Retourne VRAI si la valeur de a est strictement inférieure à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX
`a != b`	Retourne VRAI si la valeur de a est différente à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX
`a >= b`	Retourne VRAI si la valeur de a est supérieure ou égale à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX
`a <=b`	Retourne VRAI si la valeur de a est inférieure ou égale à la valeur de b, FAUX sinon	1 si VRAI, 0 si FAUX

Exercice 3.2.
Soit trois variables numériques a, b et c. Ecrire l'expression « *a est différent de b et b strictement supérieur à c ou a inférieur ou égal à c* » en langage C. Quel problème peut se poser lors de l'évaluation de l'expression ?

L'expression en langage C est : `a != b && b > c || a <= c`.

Le problème pouvant se poser est l'ordre des opérateurs à évaluer. Plusieurs interprétations sont possibles :

```
a != (b && b > c || a <= c)
```

```
a != b && (b > c || a <= c)
```

4. Priorité des opérateurs

Exercice 4.1.
Comment le langage C rend-il déterministe l'évaluation d'expression contenant plusieurs opérateurs ?

Le langage C rend déterministe l'évaluation d'une expression contenant plusieurs opérateurs en les appliquant de gauche à droite puis en donnant des priorités aux opérateurs de sorte qu'il n'existe qu'une lecture possible de l'expression.

Exercice 4.2.
Donner la table des priorités des opérateurs arithmétiques, logiques et relationnels du langage C.

La table suivante présente les opérateurs arithmétiques, logiques et relationnels du langage C par ordre de priorité descendante. Un opérateur au-dessus d'un autre est prioritaire. Des opérateurs se trouvant au même niveau sont de même priorité.

Opérateur	Associativité
`()`, `++ (suffixe)`, `-- (suffixe)`	De gauche à droite
`++ (préfixe)`, `-- (préfixe)`	De droite à gauche
`!`	De droite à gauche
`*`, `/`, `%`	De gauche à droite
`+`, `-`	De gauche à droite
`>`, `<`, `>=`, `<=`	De gauche à droite
`==`, `!=`	De gauche à droite
`&&`	De gauche à droite
`||`	De gauche à droite
`=`, `+=`, `-=`, `*=`, `/=`, `%=`	De droite à gauche

Exercice 4.3.
Soit l'expression suivante : `++a || (++b > ++c && (++d*++e))`
Si a = 1, b = 1, c = 1, d = 1 et e = 1, quelles sont les valeurs des variables après l'exécution de cette instruction ?
Que s'est-il passé durant l'exécution ?

D'après les priorités, la première valeur évaluée est ++a qui retourne `1+1 = 2`. L'expression devient alors :

```
2 || (++b > ++c && (++d*++e))
```

Une valeur de 2 représentant VRAI, d'après la table de vérité du `||`, l'expression `||` globale ne peut que valoir VRAI.

`||`	VRAI	FAUX
VRAI	1	**1**
FAUX	1	0

Il est inutile de calculer `(++b > ++c && (++d*++e))` pour évaluer l'expression globale et ++b, ++c, ++d et ++e ne sont pas exécutés.

Le programme permet d'évaluer l'expression et d'afficher les valeurs des variables après exécution :

```
#include <stdio.h>

int main()
{

    int a = 1;
    int b = 1;
    int c = 1;
    int d = 1;
    int e = 1;

    ++a || (++b > ++c && (++d*++e));

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("d = %d\n", d);
    printf("e = %d\n", e);

    return 0;
}
```

Exercice 4.4.
Soit l'expression suivante.

```
++x || (++y > z && (y*++z))
```

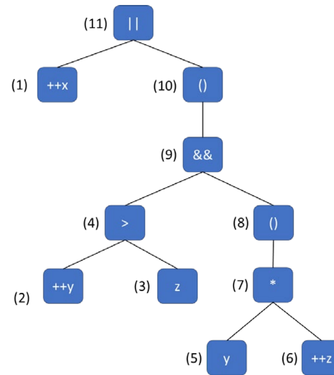
Ecrire l'arbre d'exécution de l'expression.

Numéroter les nœuds de l'arbre dans l'ordre de l'évaluation de l'instruction.

Faire apparaitre les optimisations (en retirant de l'arbre les nœuds inutilisés) si les valeurs des variables avant expression sont :

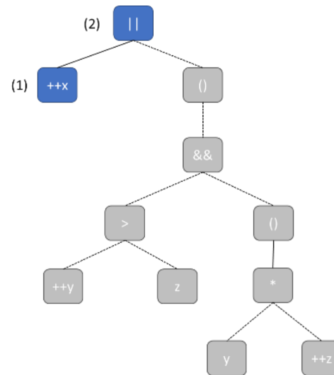
- `x = 1`, `y = 1` et `z = 1`
- `x = -1`, `y = 1` et `z = 3`
- `x = -1`, `y = 1` et `z = 0`

L'arbre de l'expression est le suivant :



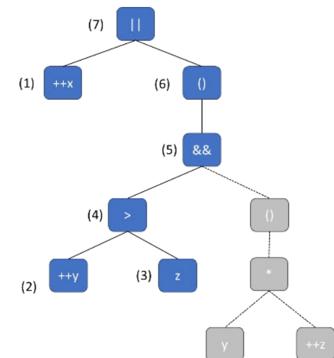
Les numéros entre parenthèse représentent l'ordre dans lequel le nœud est traité lors de l'évaluation. Ils résultent d'un parcours préfixé de l'arbre : quand on arrive sur un nœud, on parcourt d'abord tout le sous arbre gauche, puis le cous arbre droit, et enfin le nœud courant.

Les optimisations pour les valeurs initiales x = 1, y = 1 et z = 1 donnent l'arbre :



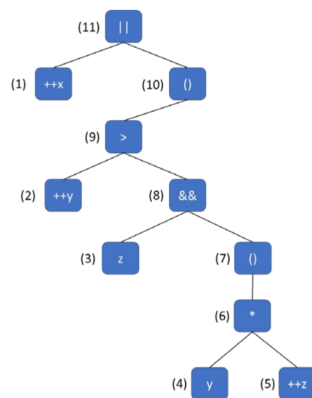
Si x est initialisé à 1, l'instruction ++x incrémente x à 2 et retourne cette valeur. Quelque-soit la valeur du reste de l'expression, le ou (2) sera égal à 1 et il n'est donc pas nécessaire d'évaluer le reste de l'expression.

Les optimisations pour les valeurs initiales x = -1, y = 1 et z = 3 donnent l'arbre :



Si x est initialisé à -1, l'instruction ++x incrémente x à 0 et retourne cette valeur. Le premier opérande du ou (7) est FAUX (0) et le reste de l'expression doit être évalué. Avec y initialisé à 1 et z initialisé à 3, même après son incrémentation (2), y est toujours inférieur à z. L'instruction ++y > z retourne FAUX (0) ce qui fait que quelque-soit la valeur du reste de l'expression, le et (5) sera évalué à FAUX (0), il 'est donc pas nécessaire d'évaluer la dernière parenthèse.

Les optimisations pour les valeurs initiales x = -1, y = 1 et z = 3 donnent l'arbre :



Il n'y a pas d'optimisation possible pour ce cas de figure.