11X001 – Introduction à la programmation des algorithmes

2.2 Expressions

François Fleuret

https://fleuret.org/francois

22 Septembre, 2020

Le contenu de ce document a été en grande partie repris du cours de Jean-Luc Falcone.

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Expressions, déclarations et commentaires

Expressions

Une expression est une combinaison d'éléments de syntaxe qui peut être évaluée en un résultat.

Le résultat produit dépend:

- des règles de priorités du langage,
- des règles d'association du langage,
- de la sémantique des opérateurs.

Expressions

Une expression est une combinaison d'éléments de syntaxe qui peut être évaluée en un résultat.

Le résultat produit dépend:

- des règles de priorités du langage,
- des règles d'association du langage,
- de la sémantique des opérateurs.

Par exemple:

```
1 + 2 * 3
```

Déclarations

Une déclaration est une combinaison d'éléments de syntaxe qui ne produit pas de résultat mais qui peut:

- Modifier le contexte de compilation: types, constantes, fonctions.
- Modifier le contexte d'exécution: contrôle du flux.
- Produire un effet de bord: affectations, entrées/sorties.

Déclarations

Une déclaration est une combinaison d'éléments de syntaxe qui ne produit pas de résultat mais qui peut:

- Modifier le contexte de compilation: types, constantes, fonctions.
- Modifier le contexte d'exécution: contrôle du flux.
- Produire un effet de bord: affectations, entrées/sorties.

Par exemple:

```
print(3)
```

Remarques

La programmation fonctionnelle évite les effets de bord.

Remarques

La programmation fonctionnelle évite les effets de bord.

Plusieurs langages mélangent expressions et déclarations, par exemple les langages dérivés de C.

E.g.

```
i = j = k++;
```

Commentaires

Les commentaires sont des annotations du programme qui sont ignorées par le compilateur.

Ils sont destinés aux humains qui liraient le code:

- Documentation.
- Clarification.
- Exemples d'utilisation.
- Références bibliographiques.

```
// Commentaire jusqu'à la fin de la ligne
1 + 2

/*  Commentaire sur
    plusieurs
    lignes */
print(3)
```

Affichage

Pour afficher une valeur, vous pouvez utiliser la fonction `print`.

C'est le seul effet de bord que nous utiliserons dans cette partie du cours.

Nous discuterons d'autres entrées/sorties dans la partie du cours consacrée à la programmation impérative.

```
print(3)     // Affiche: '3'
print(1,2,3) // Affiche: '1 2 3'
```

Expressions arithmétiques

Litéraux

Un litéral est une expression qui est directement interprétée comme valeur:

```
true  // type Bool
false // type Bool

123   // type Int
-123  // type Int

123.0 // type Double
-0.5  // type Double
1.2e3 // type Double
```

Litéraux numériques (1)

On peut obtenir un autre type numérique à partir d'un littéral en utilisant une conversion numérique:

```
UInt(123)    // type UInt
Int8(12)     // type Int8
Float(-0.5)  // type Float
```

Litéraux numériques (1)

On peut obtenir un autre type numérique à partir d'un littéral en utilisant une conversion numérique:

```
UInt(123)    // type UInt
Int8(12)     // type Int8
Float(-0.5)  // type Float
```

Les erreurs de représentation sont détectées à la compilation

```
UInt(-123)    // Erreur de compilation
Int8(2300)    // Erreur de compilation
```

Litéraux numériques (2)

On peut utiliser une base non-décimale pour définir un littéral numérique:

```
0b1101010 // Base binaire: 106 = 64 + 32 + 8 + 1
0o436     // Base octale: 286 = 4*8^2 + 3*8^1 + 6*8^0
0xBEEF    // Base hexadécimale: 48879 = 11*16^3 + 14*16^2 + 14*16^1 + 15*16^0
```

Les zéros initiaux et les underscores sont ignorés:

```
00003      // Égal à 3
1_000_000  // Égal à 1 million
5_00_000   // Égal à 1/2 million
```

Déclaration de constantes

On déclare des constantes en associant un identifiant à une expression à l'aide du mot réservé `let`.

Une assignation est une déclaration. Elle ne retourne pas de valeur.

Déclaration de constantes

On déclare des constantes en associant un identifiant à une expression à l'aide du mot réservé `let`.

Une assignation est une déclaration. Elle ne retourne pas de valeur.

```
let g = 9.8065
let three = 2 + 1
let trois = three
let weight = g * 25.0
let debugMode = true
```

Constantes: cas d'utilisation

Améliorer la lisibilité du code:

- découper une expression complexe en plusieurs parties,
- séparer les concepts et leurs valeurs,
- assurer qu'un changement est reflété dans tous le programme.

Constantes: cas d'utilisation

Améliorer la lisibilité du code:

- découper une expression complexe en plusieurs parties,
- séparer les concepts et leurs valeurs,
- assurer qu'un changement est reflété dans tous le programme.

Réutiliser le résultat d'une évaluation pour éviter de faire des calculs superflus et obtenir ainsi un programme plus efficace. *E.g.*

```
let res1 = sin(a) * cos(a) * sin(a) * cos(a)
```

est moins efficace que

```
let x = sin(a) * cos(a)
let y = x * x
```

On peut indiquer un type lors d'une déclaration de constante. *E.g.*

```
let a: Int = 12   // a est du type Int
let x: Float = 12 // x est du type Float
let b: UInt8 = 12 // b est du type UInt8
```

Inférence de type

On peut indiquer un type lors d'une déclaration de constante. *E.g.*

```
let a: Int = 12   // a est du type Int
let x: Float = 12 // x est du type Float
let b: UInt8 = 12 // b est du type UInt8
```

Sinon Swift infère le type. *E.g.*

```
let c = 12           // c est du type Int
let d = UInt8(12)    // d est du type UInt8
let debugMode = true // debugMode est un Bool
```

On obtient une erreur à la compilation si:

- on modifie la valeur d'une constante après sa déclaration, ou
- on déclare deux constantes avec le même identifiant dans le même espace de nommage.

```
let a = 12
a = 13     // error: cannot assign to value
let a = 13 // error: invalid redeclaration
```

On obtient une erreur à la compilation si:

- on modifie la valeur d'une constante après sa déclaration, ou
- on déclare deux constantes avec le même identifiant dans le même espace de nommage.

```
let a = 12
a = 13     // error: cannot assign to value
let a = 13 // error: invalid redeclaration
```

Variables

Il existe des variables en Swift, on les verra dans la troisième partie du cours.

Il n'y a pas de variable en programmation fonctionnelle.

Opérations arithmétiques

Opérateurs binaires:

- Addition `+`
- Soustraction `-`
- Multiplication `*`
- Division `/`
- Reste de division `%` (seulement entre entiers)

Opérateurs unaire:

- Opposé `-` (change le signe d'une expression)

```
let a = 100
let b = -a
```

Opérations arithmétiques: Overflow

Les opérateurs arithmétiques ne permettent pas de “déborder”, c’est à dire de dépasser les limites représentables d’un type numérique:

```
let a: Int8 = 126
a + 2   // Erreur d'execution: 'Invalid instruction'
let b: UInt64 = 0
b - 1   // Erreur d'execution: 'Invalid instruction'
```

Opérations arithmétiques: Typage fort

Les opérandes doivent être de même type:

```
let x = 1.2
let i = 2
let j = Int16(1)

x * i // Erreur de compilation
i + j // Erreur de compilation
```

Opérations arithmétiques: Typage fort

Les opérandes doivent être de même type:

```
let x = 1.2
let i = 2
let j = Int16(1)

x * i // Erreur de compilation
i + j // Erreur de compilation
```

Swift accepte de combiner des expressions litérales entières et à virgule:

```
let x = 1.2 * 2
```

Conversions numériques

On peut convertir les types numériques explicitement en utilisant le nom du type d'arrivée:

```
let x = 1.2
let i = 2
let j = Int16(1)

x * Double(i) // Resultat Double

UInt8(i) + UInt8(j) // Resultat UInt8
```

On peut convertir les types numériques explicitement en utilisant le nom du type d'arrivée:

```
let x = 1.2
let i = 2
let j = Int16(1)

x * Double(i) // Resultat Double

UInt8(i) + UInt8(j) // Resultat UInt8
```

Attention

Dans quels cas la conversion n'est pas possible ?

Que se passera-t-il dans ces cas ?

Conversions numériques

```
  1> let a = -5
a: Int = -5
  2> Int8(a)
$R0: Int8 = -5
  3> UInt8(a)
Fatal error: Negative value is not representable
  4> UInt16(65535)
$R1: UInt16 = 65535
  5> UInt16(65536)
error: repl.swift:5:8: error: integer literal '65536' overflows when
stored into 'UInt16'
```

Attention

En Swift la division entre deux entiers donne un entier:

```
let i = 12
let j = 5

print( i / j )                  // Affiche 2
print( Double(i) / j )          // Erreur de compilation
print( Double(i) / Double(j) ) // Affiche 2.4
```

En Swift la division entière par zero donne une erreur d'exécution.

Priorité des opérateurs arithmétiques (precedence)

Comme en math:

1. `*`, `/`, `%`
2. `+`, `-`

Les parenthèses changent l'ordre d'exécution

Les opérateurs de même niveau de priorité sont évalués de gauche à droite

Priorité des opérateurs arithmétiques: exemples

`a / b * c` est égal à:

Priorité des opérateurs arithmétiques: exemples

`a / b * c` est égal à:

$$\frac{a}{b}c$$

Priorité des opérateurs arithmétiques: exemples

`a / b * c` est égal à:

$$\frac{a}{b}c$$

`a / (b * c)` est égal à:

Priorité des opérateurs arithmétiques: exemples

`a / b * c` est égal à:

$$\frac{a}{b}c$$

`a / (b * c)` est égal à:

$$\frac{a}{b\,c}$$

Expressions logiques: Opérateurs booléens

Opérateurs binaires:

- Conjonction (et) `&&`
- Disjonction (ou) `||`

Opérateur unaire:

- Négation (non) `!`

```
let a = true
let b = a || false
let c = ! b
```

Expressions logiques: Table de vérité

La table suivante indique la sémantique des opérateurs booléens:

A	B	A `&&` B	A `||` B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Par exemple: `(A && B) || (!A && !B)`

Égalité

On peut tester l'égalité entre deux expressions grâce à l'opérateur: `==`

- Le résultat est du type `Bool`
- Deux expressions sont égales si leur évaluation donne le même résultat
- En principe les deux expressions doivent être de même type (sauf entiers entre eux).

L'inégalité est exprimée par l'opérateur: `!=`

Égalité: Exemple

```
(a != b) == !(a == b)  // Toujours vrai
Int(21) == UInt(21)    // Vrai
Int16(21) == Int64(21) // Vrai
21.0 == Int(21)        // Erreur de compilation
2 == true              // Erreur de compilation
0.3 == 0.2 + 0.1       // Faux
```

Attention

- Soient x et y, deux nombres à virgule flottante (`Float` ou `Double`);
- Soit ϵ un petit nombre arbitraire (généralement entre 10^{-6} et 10^{-15})
- On considère que x est égal à y si $|x - y| \leq \epsilon$.

Attention

- Soient x et y, deux nombres à virgule flottante (`Float` ou `Double`);
- Soit ϵ un petit nombre arbitraire (généralement entre 10^{-6} et 10^{-15})
- On considère que x est égal à y si $|x - y| \leq \epsilon$.

```
import Foundation // Permet d'utiliser la fonction abs

let x = 0.3
let y = 0.2 + 0.1

abs(x-y) < 1e-15   // Vrai, donc x == y
```

Comparaison

Les opérateurs suivants permettent de tester les relations d'ordre:

- `>`: plus grand
- `>=`: plus grand ou égal
- `<` : plus petit
- `<=` : plus petit ou égal

Seuls les types qui ont une relation d'ordre peuvent être comparés (par exemple: types numériques, chaînes de caractères, etc.)

Le type retourné par une comparaison est un `Bool`.

Comparaison des nombres à virgule flottante

Dans les cas des nombres à virgule flottante, les problèmes d'arrondis se posent aussi pour les comparaisons `>=` ou `<=`

```
  1> 0.2 + 0.7 - 0.9
$R0: Double = -1.1102230246251565E-16
  2> 0.2 + 0.7 >= 0.9
$R1: Bool = false
  3> 0.2 + 0.7 <= 0.9
$R2: Bool = true
```

Comparaison des nombres à virgule flottante

Dans les cas des nombres à virgule flottante, les problèmes d'arrondis se posent aussi pour les comparaisons >= ou <=

```
  1> 0.2 + 0.7 - 0.9
$R0: Double = -1.1102230246251565E-16
  2> 0.2 + 0.7 >= 0.9
$R1: Bool = false
  3> 0.2 + 0.7 <= 0.9
$R2: Bool = true
```

On ajoutera donc ϵ au nombre testé comme étant le plus petit.

```
let eps = 1e-15

x + eps < y   // Au lieu de x <= y
x  > y + eps  // Au lieu de x >= y
```

Priorité des opérateurs binaires (precedence)

1. `*`, `/`, `%`
2. `+`, `-`
3. `==`, `!=`, `>`, `<`, `<=`, `>=`
4. `&&`
5. `||`

- Les parenthèses changent l'ordre d'exécution
- Les opérateurs de même niveau de priorité sont évalués de gauche à droite

Récapitulatif

```
let i = UInt(2)
let j = 3
print( i == j )
print( i <= j  || j < i  )
print( i < 1  && j < 10 )
print( i - 2 < j - 1 )
```

Expressions avec des types agrégés

On déclare une chaîne de caractère en mettant du texte entre guillemets.

Il existe plusieurs autres manières de créer et de manipuler des chaînes de caractère. Certaines seront vues par la suite.

Les chaînes de caractères sont ordonnées par ordre lexicographique ($\simeq$ alphabétique).

```
  1> let premiere = "aaaaa"
premiere: String = "aaaaa"
  2> let seconde = "aab"
seconde: String = "aab"
  3> premiere == seconde
$R0: Bool = false
  4> premiere < seconde
$R1: Bool = true
```

Structures: instanciation

On crée une instance d'une structure en fournissant la valeur de chaque propriété.

Il faut indiquer le nom de chaque propriété et respecter l'ordre de définition.

```
struct Foo {
  let bar: Int
  let baz: Bool
}

let foo = Foo( bar: 2, baz: true )
```

Structures: instanciation (exemple)

```
struct Color {
  let red: UInt8
  let green: UInt8
  let blue: UInt8
}

struct Style {
  let foreground: Color
  let background: Color
}

let orange = Color(red:255, green:255/2, blue:0)

let pageStyle = Style(
  foreground: Color(red:0, green:0, blue:0),
  background: orange
)
```

Accès aux propriétés

On accède aux propriétés d'une structure avec l'opérateur point `.`

```
let foo = Foo( bar: 2, baz: true )

print( foo.bar )   // Affiche 2
print( ! foo.baz ) // Affiche false
```

Accès aux propriétés (exemple)

```
let orange = Color(red:255, green:255/2, blue:0)

let pageStyle = Style(
    foreground: Color(red:0, green:0, blue:0),
    background: orange
)

let gray = (orange.red + orange.green + orange.blue)/3 // *boom*

let grayStyle = Style(
   foreground: pageStyle.foreground,
   background: gray // *boom*
)
```

Par défaut, il n'y a pas de relation d'équivalence ou d'ordre entre instances.

Il faut donc traiter le cas à la main

```
style.background == style.foreground // Erreur de compilation

// Solution possible
let bg = style.background
let fg = style.foreground
bg.red == fg.red &&  bg.green == fg.green && bg.blue == fg.blue
```

On peut aussi ajouter le protocole `Equatable` à la définition de la structure.

Le compilateur synthétisera le code nécessaire pour tester l'égalité instance.

Attention aux nombres à virgule flottante...

```
struct Color: Equatable {
  let red: UInt8
  let green: UInt8
  let blue: UInt8
}

style.background == style.foreground // ça compile !
```

Tableaux: Création

- On définit un tableau en indiquant ses valeurs entre crochets.
- Tous les éléments doivent être de même type.
- Des crochets vides indiquent un tableau vide.

```
let position = [ 1.0, -2.0, 3.5 ]  // Type [Double]
let ipAddr: [UInt8] = [ 129, 194, 6, 50]
let colors = [orange, pink]  // Type [Color]
let result: [Int] = []
```

Tableaux: Accès aux valeurs

- On accède à une valeur d'un tableau en utilisant son indice.
- La première valeur a l'indice 0.
- La dernière valeur a l'indice $n - 1$ où n est la taille du tableau.
- La propriété `count` permet de connaître la taille d'un tableau.
- Si l'on essaye d'accéder à une valeur inexistante, une erreur d'exécution se produit.

```
  1> let t = [1, 2, 3]
t: [Int] = 3 values {
  [0] = 1
  [1] = 2
  [2] = 3
}
  2> print( t.count )
3
  3> print( t[0] + t[2] )
4
  4> print( t[5] )
Fatal error: Index out of range: file /home/buildnode/jenkins/workspace/oss-swif
```

Il existe de nombreuses méthodes permettant de parcourir, d'utiliser ou de modifier un tableau.

On verra certaines de ces fonctionnalités plus tard dans le semestre.

Tuples: Instanciation

- On peut instancier un tuple en listant les valeurs entre parenthèses.
- Un tuple peut être vide, ou ne comporter qu'un seul élément.
- Les éléments d'un tuple peuvent être de types différents.

```
let as = (1, 2) // Tuple (Int, Int)
let bs = (UInt8(1), Int16(2) ) // Tuple (UInt8, Int16)
let cs: (UInt8, Int16) = (1, 2) // Tuple (UInt8, Int16)
let ds = (1, true, 2.4) // Tuple (Int, Bool, Double)
let es = () // Tuple vide
let fs = (1) // Tuple (Int)
```

Tuples: Accès aux valeurs

- On peut accéder aux valeurs d'un tuple par indice en utilisant l'opérateur point, comme une propriété.
- Les indices suivent la même convention que les tableaux.
- Si on accède à une valeur inexistante, une erreur de compilation se produit.

```
let as = (12, 2, true)
print( ( as.0 > as.1 ) && as.2 ) // Affiche Vrai

let orange: (UInt8, UInt8, UInt8) = (255, 127, 0)
let gray = UInt8( ( UInt16(orange.0) + UInt16(orange.1) + UInt16(orange.2) ) / 3 )
```

Tuples: Égalité et comparaisons

- Deux tuples sont égaux s'ils ont le même types et que leurs éléments sont égaux.
- Deux tuples ont une relation d'ordre si:
  - ils ont le même type,
  - ils ont moins de 7 éléments,
  - leurs éléments sont comparables.
- Les comparaisons se font de gauche à droite (à nouveau, ordre lexicographique).

```
(1, true) != (0, false) // Vrai
(1, true) != (1, 0)     // Erreur de compilation
(0, 12) < (1, 1)        // Vrai
(1, 12) < (0, 100)      // Faux
(1, true) < (2, false)  // Erreur de compilation
```

Optionnels: Création

- On crée un optionnel:
  - soit en passant la valeur si elle existe,
  - soit en passant `nil` si elle n'existe pas.

```
let from: UInt? = 12
let to: UInt? = nil

struct Rectangle {
  let height: Double
  let width: Double
  let fillColor: Color?
}
let r = Rectangle(height:12, width:5, fillColor:nil)
```

Optionnels: Teste d'égalité

- On peut tester si un optionnel est défini en testant l'égalité à `nil`.
- Si la valeur est définie, l'optionnel sera égale à cette valeur.

```
let from: UInt? = 12
let to: UInt? = nil
print( from == nil ) // Affiche false
print( to == nil )   // Affiche true
print( from == 12 )  // Affiche true
print( from == 13 )  // Affiche false
print( to == 10 )   // Affiche false
print( to == 10.0 ) // Erreur de compilation
```

Optionnels: Déballage forcé

- Un optionnel est "emballé" (wrapped).
- Il faut donc le "déballer" (unwrap) avant de l'utiliser.
- On peut forcer le déballage avec un point d'exclamation.
- Cela produira une erreur d'exécution si l'optionnel n'est pas défini.

```
let from: UInt? = 12
let to: UInt? = nil

print( from + 2)   // Erreur de compilation
print( from! + 2 ) // Affiche 14
print( to! + 2 )   // Erreur d'exécution
```

Optionnels: Valeur par défaut

On peut également déballer une valeur optionnelle avec l'opérateur binaire `??` qui permet de passer une valeur par défaut :

```
print( (from ?? 100)+ 2 ) // Affiche 14
print( (to ?? 100)  + 2 ) // Affiche 102
```

Énumérations: instanciation

- On peut identifier les valeurs d'une énumération au moyen de la syntaxe `E.v` où `E` est l'énumération et `v` la valeur.
- La mention de l'énumération peut être ommise si son type peut être inféré

```
enum TrafficLight {
  case green, yellow, red
}

let t1 = TrafficLight.green  // Type inféré: TrafficLight
let t2: TrafficLight = .red
let t3: [TrafficLight] = [.green, .green, .yellow]
```

Énumérations: égalité

Les valeurs d'une énumération sont directement comparables (si elles n'ont pas de Tuple associé)

```
let t1 = TrafficLight.green
let t2: TrafficLight = .red
let t3: [TrafficLight] = [.green, .green, .yellow]

t1 == .green // vrai
t2 != .red   // faux
t3[1] == t1  // vrai
```

Énumérations avec tuple associé

Si l'énumération possède un tuple associé, il faut passer les valeurs de propriété à l'instanciation.

```
enum Host {
  case ipv4(UInt8, UInt8, UInt8, UInt8)
  case ipv6(UInt16, UInt16, UInt16, UInt16, UInt16, UInt16, UInt16, UInt16)
  case name(String)
}
let server1: Host = .ipv4( 129, 194, 6, 50 )
let server2: Host = .name( "www.unige.ch" )
```

Énumérations avec tuple associé: Utilisation

Nous reviendrons plus tard sur les manières d'accéder aux valeurs associées.

FIN