# Fonctions et état

Jusqu'à présent, nos programmes n'avaient aucun effet de bord.

Dès lors, la notion de *temps* n'était pas importante.

Pour tout programme qui termine, n'importe quelle séquence d'actions aurait amené au même résultat.

Cela se reflétait également dans le modèle de calcul par substitution.

Une réécriture peut être effectuée n'importe où dans un terme, et toutes les réécritures qui terminent aboutissent à la même solution.

Ceci est un résultat important du $\lambda$-calcul, la théorie sous-jacente de la programmation fonctionnelle.

Ces aspects sont étudiés plus en détail dans le cours *Fondements de la programmation*.

# Objets à état

On conçoit normalement le monde comme un ensemble d'objets, dont certains ont un état qui *change* au cours du temps.

Un objet a un état si son comportement est influencé par son histoire.

Exemple : un compte en banque a un état, car la réponse à la question

> "puis-je retirer 100 CHF ?"

peut être différente au cours de la vie du compte.

# Implantation de l'état

Toute forme d'état modifiable est construite à partir de variables.

Une définition de variable s'écrit comme une définition de valeur, mais commence par ***var*** au lieu de ***val***.

Exemple :

```
var x: String = "abc";
var count = 111;
```

Tout comme une définition de valeur, une définition de variable associe une valeur à un nom.

Toutefois, dans le cas des définitions de variables cette association peut être changée plus tard au moyen d'une affectation, qui s'écrit comme en Java.

Exemple :

```
x = "salut";
count = count + 1;
```

# État dans les objets

Les objets du "monde réel" dotés d'un état sont représentés par des objets dont certains membres sont des variables.

Exemple : Voici une classe modélisant un compte en banque.

```
class BankAccount with {
  private var balance = 0;
  def deposit(amount: Int): Unit =
    if (amount > 0) balance = balance + amount;

  def withdraw(amount: Int): Int =
    if (0 < amount && amount ≤ balance) {
      balance = balance − amount;
      balance
    } else error("fonds insuffisants");
}
```

La classe définit une variable *balance* qui contient le solde courant du compte.

Les méthodes *deposit* et *withdraw* changent la valeur du solde *balance* au moyen d'affectations.

Notez que *balance* est privée (***private***) dans la classe *BankAccount* — elle ne peut dès lors être accédée de l'extérieur de la classe.

Pour créer des comptes en banque, on utilise une notation telle que :

***val*** *myAccount* = ***new*** *BankAccount*

Le mot-clef ***new*** devant une expression de création d'instance est optionnel en Scala.

On l'utilise ici pour insister sur le fait qu'on crée un nouveau compte en banque, qui est différent des comptes créés ailleurs (voir plus loin).

Exemple : Voici une session *siris* qui manipule des comptes en banque.

```
> :l bankaccount.scala
loading file 'bankaccount.scala'
> val account = new BankAccount
val account : BankAccount = scala.interpreter.ScalaObject(...)
> account deposit 50
()
> account withdraw 20
30
> account withdraw 20
10
> account withdraw 15
java.lang.RuntimeException: fonds insuffisants
        at error(Prelude.scala:3)
        at BankAccount$class.withdraw(bankaccount.scala:13)
        at <top-level>(console:1)
>
```

L'application d'une même opération deux fois de suite à un compte produit des résultats différents. Clairement, les comptes sont des objets à état.

# Identité et changements

L'affectation pose de nouveaux problèmes pour décider si deux expressions sont "les mêmes".

Lorsqu'on exclu les affectations et écrit :

$$\textbf{val}\ x = E;\ \textbf{val}\ y = E;$$

où $E$ est une expression arbitraire, alors on peut raisonnablement admettre que $x$ et $y$ sont les mêmes. C'est-à-dire qu'on aurait aussi pu écrire :

$$\textbf{val}\ x = E;\ \textbf{val}\ y = x;$$

(Cette propriété est habituellement appelée transparence référentielle).

Mais dès qu'on autorise l'affectation, les deux formulations sont différentes. Par exemple :

$$\textbf{val}\ x = \textbf{new}\ BankAccount;\ \textbf{val}\ y = \textbf{new}\ BankAccount;$$

Q: Est-ce que $x$ et $y$ sont les mêmes ?

# Équivalence opérationnelle

Pour répondre à cette dernière question, on doit préciser ce qu'on entend par "même".

La signification précise d'être "le même" est définie par la propriété d'*équivalence opérationnelle*.

De manière quelque peu informelle, cette propriété s'énonce ainsi.

Supposons que l'on aie deux définitions $x$ et $y$.

Pour tester si $x$ et $y$ sont les mêmes, on doit :

- Exécuter les définitions suivies d'une séquence arbitraire d'opérations qui impliquent $x$ et $y$, en observant les résultats éventuels.
- Ensuite, exécuter les définitions avec une autre séquence $S$' obtenue en renommant toutes les occurrences de $y$ par $x$ dans $S$.
- Si les résultats obtenus en exécutant $S$' sont différents, alors les expressions $x$ et $y$ sont certainement différentes.

- Par contre, si toutes les paires possibles de séquences $(S, S')$ produisent le même résultat, alors $x$ et $y$ sont les mêmes.

En fonction de cette définition, voyons si les expressions

> $>$ ***val*** $x =$ ***new*** *BankAccount*
> $>$ ***val*** $y =$ ***new*** *BankAccount*

définissent des valeurs $x$ et $y$ qui sont les mêmes.

Voici une nouvelle fois les définitions, suivies d'une séquence de test :

> $>$ ***val*** $x =$ ***new*** *BankAccount*
> $>$ ***val*** $y =$ ***new*** *BankAccount*
> $>$ *x deposit 30*
> *30*
> $>$ *y withdraw 20*
> *java.lang.RuntimeException*: *fonds insuffisants*

Renommons maintenant toutes les occurrences de $y$ dans cette séquence par $x$. On obtient :

$>$ **val** $x =$ **new** $BankAccount$
$>$ **val** $y =$ **new** $BankAccount$
$>$ $x$ *deposit* *30*
*30*
$>$ $x$ *withdraw* *20*
*10*

Les résultats finaux étant différents, on conclut que $x$ et $y$ ne sont pas les mêmes.

Par contre, si on définit

**val** $x =$ **new** $BankAccount;$
**val** $y = x;$

alors aucune séquence d'opérations ne permet de différencier $x$ et $y$, donc $x$ et $y$ sont les mêmes dans ce cas.

# Création d'instances avec *new*

En Scala, on recommande d'utiliser ***new*** lors de la création d'un objet d'une classe $C$ si des applications différentes de ***new*** $C$ créent des objets qui ne sont pas les mêmes.

De manière générale, ce n'est le cas que si $C$ définit ou hérite des variables.

Par exemple, on peut toujours écrire

$$\textbf{val}\ x = \mathit{Rational}(1,\ 2);$$
$$\textbf{val}\ y = \mathit{Rational}(1,\ 2);$$

car $x$ et $y$ représentent le même nombre rationnel, malgré le fait que le nombre ait été créé deux fois.

# Affectation et modèle de substitution

Les exemples précédents montrent que notre modèle de calcul par substitution ne peut plus s'utiliser.

En effet, d'après ce modèle, on peut toujours remplacer le nom d'une valeur par l'expression qui la définit.

Par exemple, dans

***val*** $x =$ ***new*** $BankAccount;$
***val*** $y = x;$

le $x$ dans la définition de $y$ pourrait être remplacé par ***new*** $BankAccount$.

Mais nous avons vu que ce changement mène à un programme différent.

Le modèle par substitution cesse donc d'être valide dès qu'on ajoute l'affectation.

Nous verrons la semaine prochaine comment on peut modifier le modèle par substitution pour prendre en compte les affectations.

# Boucles

Postulat : Les variables permettent de modéliser l'ensemble de la programmation impérative.

Mais qu'en est-il des énoncés de contrôle comme les boucles ?

On peut les modéliser au moyen de fonctions.

Exemple : Voici un programme Scala utilisant une boucle *while* :

```
def power (x: Double, exp: Int): Double = {
  var r = 1.0;
  var i = exp;
  while (i > 0) { r = r * x; i = i - 1 }
  r
}
```

Cet exemple montre que *while* n'est pas un mot-clef.

Comment peut-on définir *while* ?

# Définition de *while*

*while* est une fonction prenant deux arguments :

- une condition, de type booléen, et
- une commande, de type *Unit*.

La condition et la commande doivent être passées par nom afin d'être réévaluées à chaque itération.

Cela nous amène à la définition suivante de *while*.

```
def while(def condition: Boolean)(def command: Unit): Unit =
  if (condition) {
    command; while(condition)(command)
  } else {
  }
```

Notez que l'appel récursif à *while* est terminal, donc l'itération n'utilise qu'une taille constante de pile.

**Exercice :** Écrivez une fonction implantant la boucle *repeat* qui doit pouvoir s'utiliser ainsi :

*repeat* {
    *commande*
} ( *condition* )

Est-il aussi possible d'obtenir la syntaxe suivante ?

*repeat* {
    *commande*
} *until* ( *condition* )

Avec l'interpréteur Scala, les fonctions *while* et *repeat* sont pré-définies dans le fichier *Prelude.scala*.

Elles sont donc disponibles pour tous les programmes.

# Boucles `for`

La boucle ***for*** de Java est une exception ; elle ne peut pas être modélisée simplement au moyen d'une fonction d'ordre supérieur.

La raison est que dans un programme du type

$$\textbf{for}\ (int\ i = 1;\ i < 3;\ i = i + 1)\ \{\ System.out.println(x);\ \}$$

les arguments ***for*** contiennent la *déclaration* de la variable $i$, qui est visible dans les autres arguments et dans le corps.

Toutefois, il existe en Scala une syntaxe similaire pour les boucles ***for***.

$$\textbf{for}\ (\textbf{val}\ i \leftarrow range(1,\ 3))\ do\ \{\ System.out.print(i + \text{" "});\ \}$$

Cela imprime *1 2 3*.
Comparez cette expression avec la suivante :

$$> \textbf{for}\ (\textbf{val}\ i \leftarrow range(1,\ 3))\ yield\ i$$
$$[1,\ 2,\ 3]$$

# Exemple étendu : simulation d'événements discrets

Examinons maintenant un exemple qui montre comment les affectations et les fonctions d'ordre supérieur peuvent être combinées de manière intéressante.

Nous allons construire un simulateur de circuits digitaux.

Cet exemple montre également comment construire des programmes de simulation d'événements discrets.

# Circuits digitaux

Commençons avec un petit langage de description de circuits digitaux.

Un circuit digital est composé de *fils* et de *composants fonctionnels*.

Les fils transportent des signaux qui sont transformés par les composants.

Nous représentons les signaux au moyen des booléens *True* et *False*.

Les composants (ou *portes*) de base sont :

- Un inverseur, dont la sortie est l'inverse de son entrée.
- Une porte ET, dont la sortie est la conjonction de ses entrées.
- Une porte OU, dont la sortie est la disjonction de ses entrées.

D'autres composants peuvent être construits en combinant ces composants de base.

Les composants ont un *délai*, donc leurs sorties ne changent pas immédiatement après un changement de leurs entrées.

# Un langage pour les circuits digitaux

Nous décrivons les éléments d'un circuit digital au moyen des classes et des fonctions Scala suivantes.

Tout d'abord, la classe *Wire* modélise les fils.

Les fils peuvent être construits ainsi :

```
val a = new Wire;
val b = new Wire;
val c = new Wire;
```

D'autre part, il existe les fonctions suivantes :

```
def inverter(input: Wire, output: Wire): Unit
def andGate(a1: Wire, a2: Wire, output: Wire): Unit
def orGate(o1: Wire, o2: Wire, output: Wire): Unit
```

qui créent les composants de base, comme effet de bord.

Des composants plus complexes peuvent être construits à partir de cela.

Par exemple, un demi additionneur se définit ainsi :

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire): Unit = {
    val d = new Wire;
    val e = new Wire;
    orGate(a, b, d);
    andGate(a, b, c);
    inverter(c, e);
    andGate(d, e, s);
}
```

Ce demi additionneur peut à son tour être utilisé pour définir un additionneur complet :

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) = {
    val s = new Wire;
    val c1 = new Wire;
    val c2 = new Wire;
    halfAdder(a, cin, s, c1);
    halfAdder(b, s, sum, c2);
    orGate(c1, c2, cout);
}
```

# Que devons-nous faire de plus ?

Pour résumer, la classe *Wire* et les fonctions *inverter*, *andGate*, et *orGate* représentent un petit langage de description de circuits digitaux.

Donnons maintenant l'implantation de cette classe et de ces fonctions qui nous permettent de simuler des circuits.

Ces implantations sont basées sur un API simple permettant la simulation par événements discrets.

# API de simulation

Un simulateur d'événements discrets effectue des actions spécifiées par l'utilisateur à un temps donné.

Une action est une fonction qui ne prend aucun paramètre et qui retourne *Unit* :

```
type Action = ()Unit;
```

Le temps est simulé ; il n'a rien à voir avec le temps de la vie réelle.

Une simulation concrète se fait à l'intérieur d'un objet qui hérite de la classe abstraite *Simulation* qui a la signature suivante :

```
class Simulation with {
  def currentTime: Int;
  def afterDelay(delay: Int)(action: Action): Unit;
  def run: Unit;
}
```

Ici,

*currentTime* retourne le temps simulé courant sous forme d'un entier.

*afterDelay* enregistre une action à effectuer après un certain délai (par rapport au temps courant *currentTime*).

*run* effectue la simulation jusqu'à ce qu'il n'y ait plus d'actions en attente.

# La classe `Wire`

Un fil doit offrir trois possibilités de base :

- *getSignal*: *Boolean* retourne la valeur actuelle du signal transporté par le fil.
- *setSignal*(*sig*: *Boolean*) modifie la valeur du signal transporté par le fil.
- *addAction*(*p*: *Action*): *Unit* attache la procédure spécifiée aux *actions* du fil. Toutes les actions attachées sont exécutées à chaque changement du signal transporté.

Voici une implantation de la classe *Wire* :

```
class Wire with {
    private var sigVal = False;
    private var actions: List[Action] = [];
    def getSignal = sigVal;
    def setSignal(s: Boolean) =
        if (s != sigVal) {
            sigVal = s;
            actions.foreach(action ⇒ action());
        }
    def addAction(a: Action) = {
        actions = a :: actions; a()
    }
}
```

L'état d'un fil est modélisé par deux variables privées :

- La variable *sigVal* représente la valeur actuelle du signal.
- La variable *actions* représente les actions actuellement attachées au fil.

# Inverseur

Nous implantons l'inverseur en installant une action sur son fil d'entrée.

Cette action produit l'inverse du signal d'entrée sur le fil de sortie.

Le changement doit être effectif après un délai de *InverterDelay* unités de temps simulé.

On obtient donc l'implantation suivante :

```
def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal;
    afterDelay(InverterDelay) {⇒ output.setSignal(!inputSig) };
  }
  input addAction invertAction
}
```

# Porte ET

La porte ET est implantée de manière similaire.

L'action d'une porte ET produit la conjonction des signaux d'entrée sur le fil de sortie.

Cela doit se produire après un délai de $AndGateDelay$ unités de temps simulé.

On obtient donc l'implantation suivante :

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {
    def andAction() = {
        val a1Sig = a1.getSignal;
        val a2Sig = a2.getSignal;
        afterDelay(AndGateDelay) {⇒ output.setSignal(a1Sig & a2Sig) };
    }
    a1 addAction andAction;
    a2 addAction andAction;
}
```

**Exercice :** Écrivez l'implantation de la porte OU.

**Exercice :** La porte OU peut également se définir en combinant des inverseurs et des portes ET. Définissez une fonction *orGate* en termes de *andGate* et *inverter*. Quel est le délai de ce composant ?

# La classe de simulation

Il ne nous reste maintenant plus qu'à implanter la classe *Simulation*.

L'idée est de garder, dans l'objet simulation, un agenda des actions à effectuer.

Cet agenda est une liste de paires. Chaque paire est composée d'une action et de l'heure à laquelle elle doit se produire.

La liste de l'agenda est triée de manière à ce que les actions à effectuer d'abord soient au début.

```
class Simulation with {

    private type Agenda = List[(Int, Action)];
    private var agenda: Agenda = [];
```

Il existe de plus une variable privée, *curtime*, qui contient le temps de simulation actuel.

```
    private var curtime = 0;
```

Une application de la méthode *afterDelay* (*delay*) (*action*) insère la paire (*curtime* + *delay*, *action*) dans la liste de l'*agenda*, à la position idoine.

Une application de la méthode *run* supprime les éléments successifs de l'agenda et effectue les actions associées.

Ce processus continue jusqu'à ce que l'agenda soit vide :

```
def run = {
    afterDelay(0){⇒ System.out.println("*** la simulation commence ***"); }
    while (!agenda.isEmpty) { next }
}
```

La méthode *run* utilise la fonction *next*, qui supprime le premier élément de l'agenda et effectue son action.

Les implantations de *next* et *afterDelay* sont laissées en exercice.

# Lancement de la simulation

Avant de lancer la simulation, il nous faut encore un moyen pour examiner les changements des signaux sur les fils.

À cette fin, nous définissons la fonction *probe*.

```
def probe(name: String, wire: Wire): Unit = {
    wire addAction {⇒
        System.out.println(
            name + " " + currentTime + " nouvelle valeur = " + wire.getSignal);
    }
}
```

Définissons maintenant quatre fils et plaçons des sondes :

```
> val input1 = new Wire;
> val input2 = new Wire;
> val sum = new Wire;
> val carry = new Wire;

> probe("somme", sum);
sum 0 nouvelle valeur = false
> probe("report", carry);
report 0 nouvelle valeur = false
```

Définissons ensuite un demi additionneur au moyen de ces fils :

```
> halfAdder(input1, input2, sum, carry);
```

Donnons maintenant la valeur *True* à *input1* et lançons la simulation :

```
> input1 setSignal True; run
*** la simulation commence ***
somme 8 nouvelle valeur = true
> input2 setSignal True; run
report 11 nouvelle valeur = true
somme 15 nouvelle valeur = false
```

etc.

# Résumé

- L'état et les affectations rendent notre modèle mental des calculs plus compliqué.
- En particulier, on perd la transparence référentielle.
- D'un autre côté, l'affectation nous permet de formuler certains programmes de manière élégante.
- Exemple : simulation d'événements discrets.
- Ici, un systèmes est représenté par une liste modifiable d'*actions*.
- Les procédures d'actions, lorsqu'elles sont appelées, changent l'état des objets et peuvent également mettre en place d'autres actions pour le futur.
- Comme toujours, le choix entre programmation fonctionnelle et programmation impérative doit se faire en fonction de la situation.