Langage et programmation

Fabien Tarissan

23 novembre 2011
(cours 1)

Plan du cours 1

1. Introduction
2. Élements du langage
   - Les expressions
   - Les instructions
   - Les fonctions
3. Premiers pas
   - L'environnement javascool
   - Des exercices
4. Un peu plus loin
   - Les fonctions récursives
   - Les tableaux

Qu'est-ce que l'informatique ?

« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes »

(Edsger W. Dijkstra)

Science du traitement automatisé de l'information

- **Science** : donc un aspect théorique (domaine des mathématiques)
- Traitement **automatisé** : c'est l'ordinateur (domaine des technologies)
- **Information** (au sens de Shannon) : tout ce qui est *numérisable* (texte, musique, voix, image, films, ADN, . . .)

Que regroupe l'informatique ?

- *Informatique théorique* : **algorithmique**, automates, logique, calculabilité, complexité, théorie des graphes, ...
- *Programmation* : différents paradigmes (**impératif**, fonctionnel, logique, orienté objet)
- ***Réseau*** : transmission de l'information (protocoles, routage, serveurs)
- *Sécurité* : cryptologie, vérification de programmes (model checking)
- ***Architecture*** *et systèmes* : code assembleur, gestion de la mémoire
- *Intelligence artificielle* : réseaux de neurones, systèmes multi-agents
- Et beaucoup d'autres : **bases de données**, traitement automatique des langues, bio-informatique, optimisation combinatoire, **robotique**, ...

L'apprentissage de la programmation

Un programme est un intermédiaire entre l'utilisateur et la machine

Besoin de choisir un **langage de programmation**, ce qui passe par les étapes de réflexion suivantes :

1. Quel paradigme ?
2. Quel langage ?
3. Quel environnement ?

Ici : choix du *paradigme impératif*, à travers l'apprentissage de (un sous-ensemble de) *Java*, en utilisant l'environnement/logiciel *javascool* :

- Éléments du langage communs à la plupart des autres langages
- Conçu pour l'apprentissage de la programmation
- Conçu pour son utilisation dans l'enseignement secondaire
- Supports existants

Bien sûr d'autres choix peuvent être faits pour apprendre à programmer (C, C++, Ada, Python, Ocaml, Scheme, ...).

Les objectifs du jour

- Comprendre comment enseigner la programmation (et pas un langage !) :
  - s'abstraire de la syntaxe
  - comprendre les mécanismes généraux (même si certains peuvent être propres à un paradigme)
- Savoir écrire et lire un programme :
  - prédire l'exécution d'un code donné
  - reconnaître la structure d'un programme
- Avoir une bonne pratique de la programmation, i.e. être capable de :
  - *documenter* et *indenter* un programme
  - préciser les *entêtes* des fonctions
  - donner le *type* des éléments manipulés
  - *tester* les programmes
- Commencer à programmer des fonctions récursives et utiliser des tableaux

Les éléments du langage

Former une expression

Une *expression* est un texte (formel) décrivant le calcul d'une *valeur*.

Une expression peut être :

- une constante : 0, 1, 0.5, `PI`, `true`, `false`, `"toto"`, ...
- une variable : c'est un identificateur, i.e. une suite de lettres et de chiffres (qui ne correspond pas à un identificateur existant) comme *x*, *y*, *ma_var1*, ...
- un opérateur ou une fonction pré-définie : $+$, $-$, $/$, $*$, sqrt, pow, ...

Un exemple : $x + (4/2)$

Il faut faire la distinction entre l'expression elle-même (*syntaxe*) et la valeur qu'elle représente (*sémantique*).

Évaluation et typage

L'*évaluation*, c'est le processus qui permet d'attribuer une valeur à une expression.

À chaque valeur correspond un *type*

Il existe donc plusieurs types. Les types simples sont :

- `boolean` pour les booléens : `true` ou `false`
- `int` pour les entiers relatifs : `-4`, `0`, `10`
- `double` pour les nombres réels (à *virgule flottante*) : `-4.0`, `3.14`
- `String` pour les chaînes de caractères : `"une chaine"`

La déclaration

```
typeVar nomVar ;
```

Cela permet de déclarer un nouvelle variable et d'associer son nom à une case mémoire de l'ordinateur :

```
boolean b;
int i;
int n;
double x;
double x,y;
String s;
```

Remarquez :

- le ; qui termine l'*instruction*
- la , qui sépare la déclaration de différentes variables de même type

L'affectation

```
nomVar = expression;
```

Cela permet de modifier la valeur d'une variable de la façon suivante :

1. on évalue l'expression `expression` en une valeur.
2. on remplace la valeur de `nomVar` par cette valeur.

```
n = 2;
n = 3*2;
n = n+1;
```

À noter que la déclaration et l'affectation peuvent être combinées :

```
double x=3.14;
String s="ma chaine";
int i=0, j=3;
```

- le signe d'égalité n'est **pas** une égalité mathématique
- il faut que les types des variables et des expressions coïncides

Expressions vs. instructions

Il est très important de distinguer l'*expression* `x+2` de l'*instruction* `y=x+2` :

- `x+2` s'évalue en une valeur
- `y=x+2` s'exécute et modifie la mémoire de l'ordinateur

Chaque instruction a un effet différent sur l'ordinateur :

- la déclaration réserve de la mémoire et l'associe à un nom
- l'affectation modifie les valeurs présentes dans la mémoire

Plus généralement, on appelle *état* de l'ordinateur l'ensemble des contenus de sa mémoire. Cette notion permet de définir formellement la *sémantique* des instructions.

La séquence

```
{ instruction1 instruction2 }
```

Cela permet de construire une nouvelle instruction à partir de 2 instructions. Plus formellement, la sémantique de cette instruction exécutée à partir d'un état e de l'ordinateur se définit par :

1. l'exécution de `instruction1` à partir de e, conduisant à l'état e'
2. puis l'exécution de `instruction2` à partir de e' conduisant à l'état e''.

Exemple : `{double x=0.0; x=3.0;}`

Par convention, la séquence

```
{ instruction1 { instruction2 { ... instructionN}...}}
```

se note simplement

```
{ instruction1 instruction2 ... instructionN }
```

Les tests

```
if (exprBooléenne) seqSiVrai else seqSiFaux
```

Le test permet de choisir une séquence d'instruction en fonction de la valeur d'une expression booléenne. L'exécution d'une telle instruction à partir d'un état *e* consiste à :

1. évaluer l'expression `exprBooléenne` dans l'état *e*
2. exécuter **une seule** des séquences en fonction de cette valeur :
   - si elle vaut `true` on exécute `seqSiVrai` dans l'état *e*
   - si elle vaut `false` on exécute `seqSiFaux` dans l'état *e*

Les expressions booléennes s'obtiennent en général à l'aide des opérateurs suivants :

- les comparateurs `==`, `!=`,`>`, `>=`,`<`,`<=`
- les opérateurs logiques : `&&` (*et*), `||` (*ou*), `!` (*non*)

Exemple : `if (x==2 && x<y)  y=0; else y=5;`

La boucle

```
while (exprBooléenne) seqInstr
```

La boucle permet de répéter un séquence d'instructions tant qu'une condition est vérifiée. L'exécution d'une telle instruction à partir d'un état e consiste à :

1. évaluer `exprBooléenne` dans l'état courant
2. en fonction de la valeur obtenue :
   - si cette valeur est `false`, l'instruction s'arrête.
   - si cette valeur est `true`, on exécute `seqInstr` dans l'état courant, ce qui conduit à un nouvel état e'. On repart ensuite à l'étape 1 avec ce nouvel état.

Exemple : `while (x<1000) x=x*2;`

Attention : avec cette instruction apparaît le problème de la non terminaison des programmes dûe aux boucles infinies.

La boucle (une variante)

```
for (instrAvant;exprBooléenne;instrApres) seqInstr
```

Cette instruction est simplement une abréviation de :

```
instrAvant
while (exprBooléenne) {
  seqInstr
  instrApres;
}
```

En général :

- `instrApres` modifie la valeur de `exprBooléenne`
- cette boucle est utilisée quand on connait le nombre de passages à faire

Exemple :

```
int i, s=0;
for(i=1; i<=10; i=i+1) s=s+i;
```

Les fonctions

Pourquoi utiliser des fonctions

Soit l'instruction `if (x<y) z=y; else z=x;` qui permet d'affecter à z la valeur du maximum de deux nombres x et y. Bien que correcte, cette instruction n'est pas très simple à utiliser. On aimerait avoir à disposition une fonction `max` qui, étant donnés 2 nombres, calcule le maximum de ces 2 nombres.

Il suffirait ensuite d'exécuter l'instruction `z=max(x,y);`

Plus généralement, une fonction permet

- d'éviter de faire appel plusieurs fois à une séquence d'instruction (*factorisation* du code)
- de résoudre un problème plus général (*abstraction* du problème)

L'entête d'une fonction

```
typeRetour nomFonction ( type1 arg1 , ... , typeN argN)
```

Comme en mathématiques, il est nécessaire de spécifier l'ensemble de départ ainsi que l'ensemble d'arrivée d'une fonction. C'est ce qu'on appelle l'*entête* d'une fonction. Elle précise :

- le nombre d'arguments
- le type de chacun de ses arguments
- le type de la valeur de retour

Dans l'exemple précédent, ce serait

```
double max ( double x ,double y)
```

La définition d'une fonction

`enteteFonction { corpsFonction }`

La définition d'une fonction consiste simplement à associer l'*entête* de fonction au *corps* de la fonction, décrit par une séquence d'instructions se terminant (généralement) par l'utilisation du mot-clef `return` permettant de renvoyer une valeur.

Dans l'exemple précédent :

```
double max (double x, double y) {
   if (x<y) return y;
   else return x;
}
```

Les commentaires

```
// ...
/* ... */
```

Les commentaires permettent de documenter le code d'un programme. Ils ne sont pas évalués ni exécutés par l'ordinateur et sont uniquement destinés à l'utilisateur. Ils sont notamment utilisés pour préciser l'utilisation d'une fonction :

- Que signifient les arguments de la fonction ?
- Que calcule la fonction ?
- Y a-t-il des hypothèses faites sur les valeurs que peuvent prendre les arguments (restriction du type déclaré) ?

Cela permet de définir un *contrat* avec l'utilisateur sur l'emploi de la fonction.

Un exemple complet

Dans l'exemple précédent, on pourrait définir la fonction `max` de la façon suivante :

```
/* Obtenir le maximum de deux nombres réels.
   x : le premier nombre
   y : le deuxième nombre
   renvoie le maximum des nombres x et y.
*/
double max (double x, double y) {
   if (x<y) return y;
   else return x;
}
```

Les étapes indispensables pour définir une fonction

1. Établir l'*entête* de la fonction et *commenter* son utilisation
2. *Inventer* la suite d'opérations à effectuer pour répondre au problème
3. *Traduire* l'algorithme en une suite d'instructions correctes et *définir* la fonction
4. *Tester* la fonction en l'appliquant sur des exemples pertinents

Quelques exercices sur ces éléments

- **fonctions de conversion** (unités de mesures, de températures, etc. . .) : permet de préciser les 4 étapes pour définir une fonction.
- **fonctions mystères** : importance du choix du nom des fonctions et de la documentation, capacité à comprendre un programme (2 étapes : prédire l'exécution d'un programme, comprendre ce qu'il fait)
- **fonctions fausses** : mise en évidence de jeux de tests, du nombre de tests nécessaires pour s'assurer que la fonction fait bien ce qu'on veut (combinatoire), comprendre les messages d'erreurs et corriger le programme
  - opération interdite (division par 0)
  - erreurs de typage
  - confusion entre = (affectation) et == (test d'égalité)
  - portée des variables (masquage et appel en dehors d'un bloc d'instructions)
- **fonctions qui ne terminent pas** : boucle infinie, importance de s'assurer (prouver) que les programmes vont finir par s'arrêter.

Première utilisation

1. Récupérer le *javascool-proglets.jar* sur le site : `http://javascool.gforge.inria.fr/index.php`
2. Lancer le fichier (cf. `http://javascool.gforge.inria.fr/index.php?page=run`)
3. Choisir un *proglet* (`abcdAlgos` par exemple)

Plusieurs possibilités s'offrent à vous :

- suivre le didacticiel (sur la droite)
- programmer :
  - possibilité de *sauvegarder* les fichiers (et donc *restaurer* les fichiers précédemment sauvegardés)
  - exécuter les programmes et étudier les résultats sur la *console* (sur la droite)
- Étudier la documentation (onglet *memo* sur la droite)

Quelques éléments d'entrée/sortie

Pour *afficher* des valeurs :

- `print(v)` : affiche dans la console la valeur de v (sans retour à la ligne)
- `print(v)` : affiche dans la console la valeur de v (avec retour à la ligne)
- `clear()` : efface ce qui est écrit dans la console

`http://javascool.gforge.inria.fr/index.php?page=api&api=./org/javascool/macros/Stdout.html`

Pour *entrer* des valeurs :

- `readInt()` : lit un nombre entier dans une fenêtre
- `readInt(s)` : lit un nombre entier dans une fenêtre en affichant la question s.
- + toutes les variantes en fonction du type nécessaire

`http://javascool.gforge.inria.fr/index.php?page=api&api=./org/javascool/macros/Stdin.html`

Les types structurés

Motivation

Les fonctions que l'on peut définir jusqu'à maintenant ne peuvent manipuler qu'un nombre pré-défini d'éléments (lié à son entête). Dès lors, si on veut implémenter une fonction calculant la moyenne d'un ensemble de notes, il nous faut une fonction calculant la moyenne de 2 notes, une autre calculant la moyenne de 3 notes, ...et ce processus n'a pas de fin.

On aimerait pouvoir disposer d'une seule fonction prenant en argument *un ensemble* de notes et faisant la moyenne de celles-ci. De telle collections d'éléments sont appelées des *structures*. L'une d'elle, très utilisée, est la structure de tableau.

Instruction de base

En tant que nouvel élément du langage, cette structure dispose :

1. d'un type : `T []` où `T` doit être remplacé par un type de base (`int`, `boolean`, ...)
2. d'une déclaration : `T [] tab = {`t_1`,` t_2`, ...,` t_n`}` qui permet de déclarer et d'initialiser un tableau de n éléments de type T, dont la première valeur est t_1, la seconde t_2, ...et la nième t_n.
3. d'un mécanisme d'affectation : `tab[i] = val` où i est un indice du tableau (un entier compris entre 0 et $n-1$).

De plus, l'expression `t.length` permet de connaître le nombre d'éléments que le tableau contient.

Par exemple : `int [] tab = { 1, 10, 100};` crée un tableau d'entier de 3 éléments :

1	10	100

L'instruction `tab[1];` permet de récupérer la valeur du tableau à l'indice 1, c'est à dire la valeur 10.

Exemples de fonctions manipulant les tableaux

Une fonction permettant l'affichage de la valeur de tableaux d'entiers :

```
/* Affiche le contenu d'un tableau d'entiers.
   tab : le tableau d'entier
*/
void affiche_tableau_int (int [] tab) {
   int i;
   for (i=0; i<tab.length; i=i+1) println(tab[i]);
}
```

Exemples de fonctions manipulant les tableaux

Une fonction permettant l'échange de 2 cases d'un tableau :

```
/* Échange la valeur de 2 cases d'un tableau.
   tab : le tableau d'entier
   i : indice de la première case
   j : indice de la seconde case
   Hypothese : les indices i et j sont valides
*/
void echange_tableau (int [] tab, int i, int j) {
   int tmp=tab[i];
   tab[i]=tab[j];
   tab[j]=tab[i];
}
```