Programmation en Caml pour Débutants

Arthur Charguéraud

6 Juin 2005

Ceci est une version linéaire imprimable du cours, mais il est fortement conseillé d'utiliser la version interactive, sur laquelle la plupart des exercices sont corrigés automatiquement. Vous la trouverez sur le site officiel du cours :

http://www.france-ioi.org/cours_caml

Pour toutes questions, suggestions, ou remarques : cours_caml@france-ioi.org

Table des Matières

Chapitre 0

Introduction

0.1 Objectifs et méthode

L'objectif de ce cours est de vous apprendre à programmer. Le cours est écrit de sorte que :

- vous n'ayez rien besoin de savoir pour arriver à tout faire,
- vous puissiez travailler de manière entièrement autonome,
- vous progressiez aussi vite que possible.

Le langage de programmation que l'on utilise ici est Objective Caml (OCAML). Lorsque vous maîtriserez ce langage, il vous sera assez facile d'en apprendre d'autres. En effet, les concepts de la programmation sont quasiment les mêmes pour tous les langages.

Ce cours :

- est facile et accessible quelque soit votre niveau :
  - aucune connaissance préalable en programmation n'est requise,
  - un niveau de collège en mathématiques est suffisant,
  - tout est expliqué en détails, il n'y a rien à deviner;
- est fait pour que vous appreniez effectivement à programmer :
  - des exemples accompagnent toujours les explications,
  - des exercices mettent systématiquement en pratique les notions évoquées,
  - les concepts abordés sont traités dans tous les détails;
- est conçu pour que vous progressiez rapidement :
  - le texte est clair et concis autant que possible,
  - les messages d'erreurs que vous obtiendrez tôt ou tard sont décortiqués,
  - des résumés des chapitres vous aide à retrouver rapidement un détail oublié.

Conseils de lecture :

- lisez tous les paragraphes, et faites tous les exercices, sans exception;
- si vous bloquez sur un exercice, cherchez au moins 10 minutes avant de regarder la correction, mais ne passez jamais plus de 20 minutes sur un petit exercice (regardez votre montre);
- lorsque vous avez fini un exercice, retenez bien la solution, elle servira presque sûrement dans la suite;
- essayez de présenter vos programmes de la même manière que dans le cours;
- évitez de faire du copier-coller de code, car c'est en écrivant du code que l'on apprend.

0.2 Qu'est-ce que programmer

Le but de ce cours est d'apprendre à programmer, c'est-à-dire être capable de contrôler la machine. Ecrire un programme, c'est donc expliquer à la machine ce qu'on veut qu'elle fasse.

Malheureusement, la machine n'est pas très intelligente, et elle ne comprend qu'un langage extrêmement limité. Et il est très pénible pour un humain d'écrire dans ce **langage machine**.

Pour parler à la machine, on va donc se faire assister par un traducteur, capable de traduire un langage évolué, dit **langage de programmation**, en langage machine. Un tel traducteur s'appelle un **compilateur**.

Le langage de programmation que nous allons utiliser s'appelle **OCaml**, on dira souvent **Caml**. Le compilateur traduit donc du **code Caml** en **code machine**.

Récapitulons les trois phases de la production d'un programme :

1. on écrit le **code source** du programme, c'est-à-dire le texte en Caml;
2. on demande au compilateur de le traduire en code machine : c'est **la compilation** du programme;
3. on demande à la machine d'effectuer le code machine : c'est **l'exécution** du programme.

0.3 Comment programmer

Dans un programme de base, il y a deux fichiers :

1. un fichier contenant le code Caml : c'est le **source du programme**.
2. un fichier contenant le code machine : c'est **l'exécutable**.

Par exemple, si on choisit d'appeler `"monsource.ml"` le source et `"monprog.exe"` l'exécutable, les trois phases de production du programme seront :

1. éditer le fichier `"monsource.ml"` pour y écrire le source Caml
2. effectuer la traduction à l'aide de la commande `"ocamlc -o monprog.exe monsource.ml"`
3. exécuter le programme; pour cela, on tape simplement son nom : `"monprog.exe"`

Petite explication : le nom du compilateur est `ocamlc`, le petit `-o` signifie `output`, et permet de choisir le nom du programme qui va être généré.

0.4 Dialoguer avec un programme

Que peut faire un programme lorsqu'il est exécuté ? A peu près ce qu'il veut, tant que ce n'est pas interdit par le système. Pour faire quelque chose d'intéressant, le programme doit communiquer. S'il reste isolé, il ne pourra pas produire quoi que ce soit. Voici les trois moyens de communication qu'a un programme :

1. communiquer avec l'utilisateur,
2. communiquer avec des fichiers,
3. communiquer avec d'autres programmes.

Nous nous intéresserons surtout à la première partie. On verra à la fin comment lire et écrire des fichiers. On ne s'occupera pas de la communication entre les programmes, car c'est un peu technique et que ça dépend du système d'exploitation.

Pour communiquer avec l'utilisateur, il y a deux interfaces possibles.

1. On peut utiliser une interface graphique avec des menus, des boutons, des fenêtres, des boîtes de dialogues, etc... mais c'est assez pénible à programmer, et ça n'est intéressant que pour des logiciels destinés à être utilisé par beaucoup de gens.
2. On peut utiliser une interface texte. Dans ce cas, le programme ouvre une fenêtre destinée à contenir du texte.
   - Le programme peut afficher du texte dans la fenêtre. On dit alors que le programme écrit sur **la sortie standard**, et que l'utilisateur lit sur cette sortie standard.
   - L'utilisateur peut taper du texte dans la fenêtre et valider avec la touche entrée, et envoyer ainsi ce texte au programme. On dit alors que l'utilisateur écrit sur **l'entrée standard**, et que le programme lit sur cette entrée standard.

0.5 Interface de développement

Les trois phases du développement sont, on le rappelle, éditer le source, compiler le programme, et enfin l'exécuter. **Une interface de développement intégré (IDE)** est un environnement qui permet de travailler sur ces trois phases de manière efficace. Pour ce cours de programmation, vous avez le choix entre deux types d'IDE.

A) Vous pouvez utiliser un logiciel spécialisé qui fonctionne de la manière suivante :

1. C'est un véritable éditeur de fichiers texte, numérotant les lignes, capable de mettre en couleur les mots clés du langage, permettant d'annuler ou de rétablir des modifications, possédant des fonctions de recherche et de remplacement, avec des dizaines d'outils configurables, etc... bref, tout ce dont vous pourrez avoir besoin.
2. Vous pouvez compiler le programme avec la combinaison des touches "Contrôle" et "1" (on note CTRL+1). Les messages données par le compilateur sont alors affichés dans une zone prévu à cette effet, ce qui permet d'avoir en parallèle le source et la liste des erreurs trouvées dans ce source.
3. Vous pouvez exécuter le programme avec la combinaison des touches "Contrôle" et "2" (CTRL+2). Une fenêtre s'ouvre alors pour dialoguer avec le programme. Lorsque le programme se termine, on retourne immédiatement à l'éditeur.

Un tel logiciel s'avère indispensable dès que l'on veut faire de la programmation un peu sérieusement. Notez qu'il faut que Objective Caml soit installé en plus du logiciel. Un document disponible sur le site de France-IOI explique comment installer et configurer correctement ces outils.

B) Vous pouvez utiliser l'interface Web-IDE : il s'agit d'un IDE directement intégré dans la version Web de ce cours de programmation. Dans ce cas, il n'y a rien besoin d'installer sur votre machine, tout peut se faire via votre navigateur internet, sur le site de France-IOI.

1. Une zone de texte vous permet de saisir votre source. Il s'agit donc d'un éditeur relativement sommaire, mais suffisant pour écrire ses premiers programmes.
2. Vous pouvez tester votre programme en appuyant sur un bouton. Si le source ne compile pas correctement, les erreurs sont affichées, sinon, le programme est compilé puis exécuté.
3. Du fait que l'exécution se déroule à distance, il faut impérativement fournir les données à envoyer sur l'entrée standard à l'avance. Cela veut dire que vous devez taper à l'avance tout le texte que vous voulez fournir à votre programme durant son exécution. Cela peut poser des problèmes avec des programmes où vous souhaitez justement fournir à votre programme du texte différent selon les informations qu'il vous affiche. En bref, certains problèmes ne pourront pas être testé convenablement avec le Web-IDE.
4. Le Web-IDE dispose d'un bouton permettant de sauvegarder le source sur lequel vous travaillez, et d'un autre permettant de récupérer la dernière sauvegarde effectuée. Sauvegarder régulièrement vous permet d'éviter de perdre des données à cause d'une erreur de manipulation sur votre navigateur.

Le Web-IDE est donc une solution pratique pour débuter, puisqu'il permet de réaliser ses premiers programmes sans avoir besoin d'installer ou de configurer des logiciels.

0.6 Correction automatique

Qu'elle que soit l'interface de développement que vous choisissez d'utiliser, vous pouvez faire corriger vos programme en ligne avec la version Web de ce cours.

1. Copiez le source de votre programme dans la zone d'édition de l'exercice que vous souhaitez faire corriger.
2. Appuyez sur le bouton de soumission pour lancer l'évaluation du source.
3. Après quelques instants, le résultat de l'évaluation s'affiche.
4. Si vous avez réussi, la correction de l'exercice est affiché en même temps.
5. Si vous n'avez pas réussi, il vous est affiché les détails d'un exemple sur lequel votre programme n'a pas réagit comme il aurait du.
6. Si vous n'arrivez pas à résoudre le problème, un bouton vous permet d'obtenir la correction.
7. Vous pouvez récupérer par la suite le source et le résultat de la dernière évaluation faîte pour un exercice.

Le fonctionnement du correcteur automatique est très simple. On donne à votre programme et au programme de la correction la même entrée, et on regarde s'ils donnent tous les deux la même sortie. Si les sorties sont différentes, cela veut dire que votre programme ne fait pas exactement ce qui est demandé par l'exercice. Ce système est peu tolérant : une majuscule à la place d'une minuscule suffit à faire rejeter votre solution. C'est fait exprès, car pour programmer bien, il faut s'entraîner à être rigoureux.

Attention, le correcteur n'est pas toujours parfait. En particulier :

- le correcteur automatique ne corrige pas certains exercices; dans ce cas, l'exercice est considéré comme résolu dès que vous effectuez une soumission.
- c'est assez rare, mais il peut arriver que le correcteur automatique juge votre programme faux juste parce que vous ne présentez pas la sortie comme le programme de correction; il ne faut pas insister si vous estimez que votre programme est correct.

Globalement, le correcteur automatique permet une évaluation correcte d'une très grande majorité des exercices.

Have fun !

Chapitre 1

let, if, for

1.1 Instructions

1.1.1 Afficher du texte

▶ On va commencer par écrire un programme qui affiche une phrase, par exemple `"Bonjour !"`. En Caml, on va utiliser la commande `print_string`. `print` se traduit `affiche` et `string` signifie `texte` en programmation. Voici notre premier programme :

```
print_string "Bonjour !";
```

Regardons cela de plus près :

- on commence avec la commande `print_string`,
- on place un espace,
- on écrit, entre guillemets, le texte à afficher,
- on place un point-virgule pour dire que l'on a terminé une instruction.

Exécutez ce programme, et vous obtiendrez :

```
Bonjour !
```

▶ Il est possible d'effectuer plusieurs instructions de suite dans un programme.

Pour effectuer plusieurs instructions, il suffit de les écrire à la suite les unes des autres.

Écrivons un programme qui affiche d'abord `"Bonjour !"`, et ensuite `"Au revoir !"` sur la ligne d'en dessous. L'instruction `print_newline();`, traduire `"affiche une nouvelle ligne"`, permet de passer à la ligne. Attention, le couple de parenthèses à la fin y est indispensable, pour une raison que l'on comprendra plus tard.

On va donc utiliser trois instructions :

- une première pour afficher `"Bonjour !"`,

- une deuxième pour passer à la ligne,
- une troisième pour afficher `"Au revoir !"`,

Le code du programme est donc :

```
print_string "Bonjour !";
print_newline();
print_string "Au revoir !";
```

Il affiche bien

```
Bonjour !
Au revoir !
```

► Le compilateur ne tient pas compte des retours à la ligne dans le source du programme qu'on lui donne. En théorie, on aurait tout à fait pu écrire tout sur une seule ligne :

```
print_string "Bonjour !"; print_newline(); print_string "Au revoir !";
```

En pratique, l'objectif est d'écrire des programmes sans faire d'erreurs. Pour détecter les erreurs éventuelles, il faut pouvoir relire le code facilement, et pour cela, il faut qu'il soit présenté joliment. Pour cette raison, on adoptera la convention suivante, à suivre impérativement :

On écrira une seule instruction par ligne.

1.1.2 Afficher des nombres

► Pour afficher un nombre entier, on utilise la commande `print_int`, se traduisant littéralement `affiche_entier`. Par exemple `print_int 12;` permet d'afficher `"12"`.

Essayer de prévoir ce que le code qui suit va afficher :

```
print_string "Voici dix-huit : ";
print_newline();
print_int 18;
print_newline();
```

Vous avez trouvé ? Alors lancez le programme, et vérifiez qu'il vous affiche ceci :

```
Voici dix-huit :
18
```

★ Ecrivez maintenant un programme qui affiche d'abord l'entier 5, puis le texte `"a pour carré"`, et ensuite l'entier 25. Attention, il faut mettre des espaces entres les nombres et le texte.

_ _

Si on code :

```
print_int 5;
print_string "a pour carré";
print_int 25;
```

Alors on obtient comme résultat :

```
5a pour carré25
```

Ceci est du au fait que la fonction `print_int` n'affiche aucun espace avant ou après l'entier. Pour obtenir un résultat satisfaisant, il faut donc rajouter des espaces, avant et après le texte que l'on affiche avec `print_string` :

```
print_int 5;
print_string " a pour carré ";
print_int 25;
```

Et là c'est nettement mieux :

```
5 a pour carré 25
```

▶ Bien sûr, on aurait pu écrire le même programme beaucoup plus simplement, et faisant :

```
print_string "5 a pour carré 25";
```

Alors quel est l'intérêt du `print_int` ? Eh bien déjà lorsqu'on effectue un calcul et qu'on veut en afficher le résultat. Par exemple :

```
print_int 267;
print_string " a pour carré ";
print_int (267 * 267);
```

Caml comprend les expressions mathématiques simples, avec des parenthèses et des opérations de bases.

1.1.3 Erreurs

▶ On va maintenant simuler des erreurs de programmation dans les codes précédents. Le but est d'être capable de déchiffrer les messages d'erreurs du compilateur, car vous ferez forcément un jour ou l'autre des petites erreurs dans votre code. Bien lire les parties sur les messages d'erreur permet ensuite de gagner beaucoup de temps lorsque vous programmez.

▶ Simulons d'abord l'oubli d'un point-virgule :

```
print_int 5
print_newline();
```

On obtient le message suivant (en supposant que le fichier source est `test.ml`) :

```
File "test.ml", line 1, characters 0-9:
This function is applied to too many arguments,
maybe you forgot a ';'
```

D'abord on remarque que le compilateur est subtil sur un cas aussi simple, puisqu'il suggère l'oubli possible d'un point-virgule : `"maybe you forgot a ';'"`. Mais ce ne sera pas toujours le cas, et souvent il faudra se contenter de la phrase précédente : `"This function is applied to too many arguments"`, traduction : `"Cette fonction est appliquée avec trop d'arguments"`. De quelle fonction parle le compilateur ? C'est

indiqué juste au-dessus : il s'agit de la fonction dont le nom est compris entre les caractères 0 et 9 de la ligne numéro 1. En regardant le code, on s'aperçoit que c'est `"print_int"`. `"print_int"` s'utilise normalement en écrivant simplement un entier derrière. Ici, on a écrit d'autre choses après le 5 dans l'expression : `print_int 5 print_newline()`. D'où le message disant que trop d'arguments ont été fournis.

▶ Testons maintenant un autre oubli, celui de parenthèses après `print_newline()`.

```
print_string "Voici dix-huit : ";
print_newline;
print_int 18;
```

On obtient le message suivant :

```
File "test.ml", line 2, characters 0-13:
Warning: this function application is partial,
maybe some arguments are missing.
```

Un avertissement nous est donné : la fonction `print_newline` est appliquée `"partiellement"`. Lorsqu'on étudiera les fonctions, on comprendra la signification exacte du `"partiellement"`, et on verra pourquoi les parenthèses `()` sont appelées `"arguments"`. Pour l'instant, retenons simplement qu'il s'agit d'un oubli de quelque chose à la suite du nom de la fonction, ici `print_newline`.

Comme il ne s'agit que d'un avertissement, le programme fonctionne tout de même. Seulement il ne fait pas ce qu'on l'on souhaitait, puisque aucun retour à la ligne n'est effectué :

```
Voici dix-huit : 18
```

▶ Étudions encore un petit problème courant avec les nombres négatifs :

```
print_int -16;
```

Donne :

```
File "test.ml", line 1, characters 0-9:
This expression has type int -> unit but is here used with type int
```

On expliquera dans le chapitre suivant que ce message veut dire que le compilateur essaie d'effectuer la soustraction entre `print_int` et `16`, et qu'il n'y arrive pas car `print_int` n'est pas un entier. Ce qu'il faut retenir, c'est que la version correcte utilise des parenthèses :

```
print_int (-16);
```

Pour la même raison, il faudra ajouter des parenthèses autour des nombres négatifs pour effectuer des calculs :

```
print_int (-267);
print_string " a pour carré ";
print_int ((-267) * (-267));
```

1.1.4 Déclaration

▶ Considérons le programme suivant, qui affiche 5, puis le texte `"a pour carré"`, et enfin le carré de 5. Notez qu'on ne va pas afficher 25 cette fois, mais bien le résultat du calcul de 5 fois 5. Pour utiliser `print_int`

correctement, il faut le faire suivre d'un entier, et rien d'autre. Si on veut afficher le résultat de `5 * 5`, il est impératif de le mettre entre parenthèses. Comme en mathématiques, les parenthèses les plus imbriquées sont calculées en premier.

```
print_int 5;
print_string " a pour carré ";
print_int (5 * 5);
```

On aimerait maintenant faire la même chose pour 238, afin d'afficher la phrase indiquant sa valeur et celle de son carré. Il suffit pour cela de remplacer 5 par 238.

```
print_int 238;
print_string " a pour carré ";
print_int (238 * 238);
```

► Si on voulait recommencer avec d'autres valeurs, on serait obligé de faire 3 remplacements à chaque fois, ce qui risque d'être un petit peu pénible. On va donc présenter un procédé permettant, un peu comme en mathématiques, de poser `x` la valeur pour laquelle on veut effectuer les trois lignes de code précédentes. Une fois la valeur de `x` posé, la lettre `x` désignera cette valeur. Voici la démarche à suivre :

```
let x = (...la valeur de x...) in (...instructions utilisant x...)
```

Décryptons cela :

1. `"let"` se traduit `"posons"` ou encore `"soit"`,
2. `"x"` est le nom de la valeur que l'on est en train de poser,
3. le signe `=` indique qu'on va en préciser la valeur,
4. `"in"` se traduit `"dans"` ou `"à l'intérieur de"`, (sous-entendu des instructions suivantes).
5. il n'y a pas de point-virgule après le `in`.

Concrétisons cette théorie dans l'exemple qui nous intéresse :

```
let x = 238 in
print_int x;
print_string " a pour carré ";
print_int (x * x);
```

la valeur 238 n'apparaît plus qu'une seule fois, et on peut maintenant la changer en un seul remplacement

La règle d'exécution d'un `"let nom = ... in"` est la suivante :

1. calcule la valeur de l'expression entre le signe `=` et le `in`,
2. remplace le nom par cette valeur dans les instructions concernées,
3. passe alors à l'exécution de la suite.

Remarque : on précisera plus tard les notions générales de **valeurs** et **d'expressions**.

Vérifiez qu'en appliquant la règle pas à pas, le code utilisant `let x = 238` est équivalent du point de vue du comportement à celui d'origine :

```
print_int 238;
print_string " a pour carré ";
print_int (238 * 238);
```

En plus, on n'a plus de problème de parenthèses manquante avec les nombres négatifs, comme le montre l'exemple suivant :

```
let x = -12 in
print_int x;
print_string " a pour carré ";
print_int (x * x);
```

▶ On a utilisé le nom `"x"` pour y associer des valeurs. On aurait très bien pu prendre comme nom n'importe quel mot satisfaisant à la condition suivante :

Un nom de valeur doit être formé de lettres de bases (de a à z ou de A à Z), de chiffres, et d'underscores (le symbole _), et doit en outre commencer par une lettre minuscule. Il ne faut pas mettre d'espaces, ni de lettres accentuées ou tout autre caractère spécial.

★ Ecrivez un programme qui pour un nombre `x` égal à 42, en affiche la valeur, puis son double, et enfin son triple. Les valeurs affichées devront être séparées par des espaces.

- -

```
let x = 42 in
print_int x;
print_string " ";
print_int (2 * x);
print_string " ";
print_int (3 * x);
```

▶ Attention, après un `in`, il doit toujours y avoir au moins une instruction. Écrire juste :

```
let x = 42 in
```

```
File "test.ml", line 1, characters 13-13:
Syntax error
```

entraîne un message d'erreur sur le `in`. La raison est qu'il ne sert à rien de faire une déclaration si on n'en fait rien du tout après.

1.2 Interventions

1.2.1 Lecture d'un entier

▶ Nous allons maintenant voir un cas où l'utilisation du `let` est indispensable. Il s'agit de demander un entier à l'utilisateur lors de l'exécution du programme, et d'afficher sa valeur et celle de son carré. La requête d'un nombre entier à l'utilisateur se fait avec la commande `read_int()`, traduction `"lire un entier"`. Attention, les parenthèses dans `read_int` sont indispensables, de la même façon qu'elles l'étaient pour le `print_newline()`. En choisissant de poser `mon_nombre` la valeur fournie par l'utilisateur, on écrira pour nommer cette valeur :

```
let mon_nombre = read_int() in
```

Donc l'ensemble du code que l'on veut écrire est :

```
print_string "Entrez le nombre : ";
let mon_nombre = read_int() in
print_int mon_nombre;
print_string " a pour carré ";
print_int (mon_nombre * mon_nombre);
```

Pour comprendre ce qui se passe lors de l'exécution du code :

Un `read_int()` bloque l'exécution du programme jusqu'à ce que l'utilisateur ait tapé un nombre entier, et ait appuyé sur la touche entrée du clavier. A ce moment le mot `read_int()` est remplacé par la valeur donnée.

D'abord le programme affiche `"Entrez le nombre :  "`. Ensuite il s'arrête, et attend que l'utilisateur tape un nombre. Supposons que celui-ci tape 15, puis la touche entrée. Le fait d'appuyer sur entrée provoque un retour à la ligne dans l'écran d'affichage, bien qu'il n'y ait pas d'instruction `print_newline()` utilisée. A ce moment-là, le code restant est équivalent à :

```
let mon_nombre = 15 in
print_int mon_nombre;
print_string " a pour carré ";
print_int (mon_nombre * mon_nombre);
```

La règle du `let` fait que `mon_nombre` est remplacé par sa valeur dans les instructions suivantes, et il reste alors :

```
print_int 15;
print_string " a pour carré ";
print_int (15 * 15);
```

Finalement, sur l'écran final, il sera affiché :

```
Entrez le nombre : 15
15 a pour carré 225
```

▶ Faites aussi un test avec un nombre entier négatif, par exemple `-8`. Vous devriez avoir au final :

```
Entrez le nombre : -8
-8 a pour carré 64
```

1.2.2 Exercices

★ Ecrivez un programme qui affiche le texte `"Veuillez entrer un nombre :  "`, qui demande ensuite un nombre entier à l'utilisateur, et qui affiche alors sur la ligne suivante `"Votre nombre est :  "`, suivi de la valeur du nombre.

– –

```
print_string "Veuillez entrer un nombre : ";
let mon_nombre = read_int() in
print_string "Votre nombre est : ";
print_int mon_nombre;
```

★ Écrire un programme qui demande à l'utilisateur deux nombres, puis qui en affiche la somme.

_ _

```
print_string "Donnez le premier nombre : ";
let x = read_int() in
print_string "Donnez le second nombre : ";
let y = read_int() in
print_string "Leur somme vaut : ";
print_int (x + y);
```

Comme on a maintenant compris comment fonctionnait l'intervention de l'utilisateur, on va pouvoir s'affranchir de l'affichage de texte décoratif. Cela permettra d'écrire plus rapidement les programmes. Ainsi pour cet exercice, on fera :

```
let x = read_int() in
let y = read_int() in
print_int (x + y);
```

Remarque : il est aussi possible de poser le résultat du calcul de la somme avant de l'afficher.

```
let x = read_int() in
let y = read_int() in
let somme = x + y in
print_int somme;
```

1.2.3 Erreurs

► Tout d'abord une petite erreur d'exécution : un `read_int()` demande un entier à l'utilisateur. Que se passe-t-il si celui-ci tape autre chose qu'une valeur entière ? Testons cela :

```
let x = read_int() in
print_int x;
```

Vous pouvez essayer de taper des lettres (comme `"blabla"`), ou bien un nombre réel (par exemple `"3.14"`), ou n'importe quelle autre séquence de caractères qui ne représente pas un entier, et vous obtiendrez l'affichage :

```
Fatal error: exception Failure("int_of_string")
```

Il y a eu une `"Erreur fatale"` dans le traitement nommé `"int_of_string"`, traduire `"Nombre entier à partir d'un texte"`. Cela veut simplement dire que le programme n'a pas su comment traduire le texte que l'utilisateur a donné en un nombre entier correct.

► Une erreur fréquente est la mise d'un point-virgule à la place d'un `in`. Pour des raisons que l'on comprendra plus tard, le compilateur n'est pas capable de détecter proprement une telle erreur. Le message d'erreur obtenu est variable selon les cas.

Lorsque l'erreur est localisée sur la ligne du `let` en question, on s'aperçoit rapidement du problème. Mais ce n'est pas toujours le cas, et il est tout à fait possible que le compilateur indique une erreur de syntaxe plus loin dans le source. Lorsqu'on obtient une erreur de syntaxe déroutante, le mieux est de relire tranquillement le code du début à la fin, en vérifiant les `in` et les points-virgules.

Voici un cas possible :

```
let x = 5;
print_int x;
```

```
File "test.ml", line 1, characters 8-9:
Warning: this expression should have type unit.
File "test.ml", line 2, characters 10-11:
Unbound value x
```

On n'a pas les moyens de trop comprendre la première phrase, mais la seconde est claire : `Unbound value x` qui se traduit : `x` n'est pas défini. On en déduit qu'il doit y avoir une erreur dans sa déclaration.

▶ Dans `read_int()`, et aussi dans `print_newline()`, il n'y a pas besoin d'espace entre le mot et le couple de parenthèses. Toutefois, ce n'est pas une erreur d'en mettre un comme dans `"read_int ()"`. Par contre l'oubli du couple de parenthèses à la fin de `read_int()` est une erreur :

```
let mon_nombre = read_int in
print_int mon_nombre;
print_newline();
```

```
File "test.ml", line 2, characters 10-20:
This expression has type unit -> int but is here used with type int
```

Contrairement aux messages d'erreurs que nous avons vues auparavant, il ne s'agit pas d'une erreur de syntaxe, mais d'une erreur de type. Le type d'une valeur, c'est sa nature. Ainsi le type de 5 est `"entier"`, ce qui se dit `int` (comme abréviation de `integer`). Le texte `"blabla"` est de type `"string"` (pour `"chaîne de caractère"`). Il y a beaucoup d'autres types, et on aura l'occasion d'en voir d'autres plus tard. Essayons pour l'instant d'apprendre une méthode permettant d'analyser les messages d'erreurs de type.

L'erreur se situe à la ligne 2, sur le texte `mon_nombre`. Le compilateur nous dit qu'il pensait que `mon_nombre` était du type `unit -> int`, que l'on lit `"unit donne int"`. Cela n'est pas en accord avec l'utilisation que l'on souhaite faire de `mon_nombre` à la ligne 2 : puisque `print_int` permet d'afficher un entier, `mon_nombre` devrait être de type `int` (entier). Nous ne sommes pas à ce point en mesure de comprendre le sens de `unit -> int`, mais ce dont nous sommes sûrs, c'est que ce n'est pas le type `int`. On comprend qu'il y a donc une erreur dans la définition de `mon_nombre`, ce qui nous permet de retrouver l'oubli des parenthèses.

▶ On a dit tout à l'heure que l'utilisation du `let` était obligée lors du `read_int`, car on avait besoin d'utiliser la valeur donnée par `read_int()` plusieurs fois. Lorsqu'on ne souhaite l'utiliser qu'une seule fois, il est possible de l'utiliser directement. Ainsi le code suivant demande un entier à l'utilisateur, et l'affiche tout de suite après ;

```
print_int (read_int());
```

Mais cette présentation réduit la lisibilité, et pourra en plus poser des problèmes par la suite, donc on appliquera la règle :

Pour demander des données à l'utilisateur, on en posera toujours la valeur avec un `let`.

1.2.4 Lecture de texte

▶ On sait demander un entier à l'utilisateur. Voyons maintenant comment lui demander du texte. Pour demander à l'utilisateur une ligne de texte, on utilise la commande `read_line()`, traduction `"lire une ligne"`. Cette commande s'utilise de la même manière que `read_int()`.

`read_line()` permet de demander à l'utilisateur une ligne de texte.

★ Ecrivez donc un programme qui affiche `"Tapez votre texte : "`, qui demande alors un texte à l'utilisateur, puis qui affiche le texte fourni sur la ligne suivante.

_ _

```
print_string "Tapez votre texte : ";
let text = read_line() in
print_string text;
```

Remarque : on peut se demander pourquoi la commande n'est pas nommée `read_string()`. Comme on le verra plus tard, une `string` peut contenir plusieurs lignes de textes, alors que la commande qu'on utilise ici ne permet de lire qu'une seule ligne à la fois. En effet, pour écrire plusieurs lignes, il faudrait utiliser la touche entrée, mais ici cette touche sert à terminer la saisie du texte.

1.3 Conditions

1.3.1 Le `if`

► Nous allons voir une structure qui permet de déterminer le comportement d'un programme en fonction du résultat d'un test. Pour commencer, on va essayer d'écrire un programme qui demande à l'utilisateur un nombre entier, et qui affiche `"positif"` ou `"négatif"` en fonction du signe de ce nombre. La structure que nous allons utiliser est le `if, then, else`, traduire : `"si, alors, sinon"`.

Dans la description du schéma qui suit, on utilise le mot `"bloc"` pour signifier `"un groupe d'instructions"`.

```
if (...la condition...) then
   (...bloc du then...)
else
   (...bloc du else...)
;
```

Fonctionnement de la structure `if, then, else` :

- On teste la condition qui se trouve après le `if`.
- Si ce test est vérifié, on exécute juste le bloc du `then`.
- Dans le cas contraire, on exécute juste le bloc du `else`.
- Dans les deux cas, on continue ensuite l'exécution après le point-virgule finale.

Utilisation du `if, then, else` :

1. La condition peut s'exprimer avec des comparaisons de valeurs : `=` pour l'égalité, `<>` pour la différence, `<` et `>` pour les comparaisons strictes, et `<=` et `>=` pour les comparaisons larges.
2. Afin de délimiter les blocs, chacun d'entre eux commence par `begin` et se termine par `end`. Le code de chaque bloc doit, pour des raisons de lisibilité, être indenté (décalé vers la droite).

3. La totalité de la structure `if .. then .. else ..` constitue une grosse instruction, et il faut le terminer par un point-virgule.

Maintenant on peut écrire le programme qui affiche le signe d'un entier demandé par l'utilisateur (on considère ici zéro comme positif).

```
let mon_nombre = read_int() in
if mon_nombre >= 0 then
   begin
   print_string "positif";
   end
else
   begin
   print_string "négatif";
   end
;
```

Remarque : on peut aussi tester la condition inverse, c'est-à-dire déterminer si le nombre est strictement négatif, ce qui donne :

```
let mon_nombre = read_int() in
if mon_nombre < 0 then
   begin
   print_string "négatif";
   end
else
   begin
   print_string "positif";
   end
;
```

► On verra un peu plus tard qu'il est parfois possible de ne pas écrire explicitement les bornes `begin` et `end`. Pour l'instant, on les mettra systématiquement.

1.3.2 Portée

► Les déclarations réalisées jusqu'à maintenant consistaient à associer un nom à une valeur dans toute la suite du programme. Ce cas n'est pas général : il arrive qu'un nom soit associé à une valeur uniquement durant un petit morceau du programme. On dit alors que la `"portée"` de la déclaration est limitée. On va tout de suite comprendre pourquoi toutes les portées ne sont pas infinies, à l'aide de l'exemple suivant (attention, ce code n'est pas valide, on explique pourquoi juste après) :

```
let mon_nombre = read_int() in
if mon_nombre >= 0 then
   begin
   let a = 5 in
   end
else
   begin
   let b = 8 in
   end
;
print_int a;
```

Si ce code était valide, lorsque `mon_nombre` est positif, la valeur de `a` c'est-à-dire 5, sera affiché. Mais dans le cas contraire, le bloc du `else` est effectué, et le nom `a` n'est associé à rien. Que devrait dont afficher `print_int a` dans un tel cas ? Faut-il mieux afficher zéro, ou rien du tout, ou encore provoquer une erreur ? Afin de minimiser les risques d'erreurs et d'éviter tout comportement anormal dans un programme, on va tout simplement empêcher d'utiliser des noms dont on n'est pas sûr qu'ils aient été associés. D'où la règle suivante :

Une association effectuée dans un des blocs d'une structure en `if` a une portée limitée à ce bloc.

Ainsi le code suivant est tout à fait correct, mais après le premier `end`, le nom `a` n'est plus associé à rien, et après le second `end`, `b` n'est plus associé à rien :

```
let mon_nombre = read_int() in
if mon_nombre >= 0 then
   begin
   let a = 5 in
   print_int a;
   end
else
   begin
   let b = 8 in
   print_int b;
   end
;
```

1.3.3 Erreurs

► Avant d'attaquer les exercices, regardez rapidement le message d'erreur obtenu lorsqu'on oublie un `then`. Voici un tel exemple :

```
let mon_nombre = read_int() in
if mon_nombre >= 0
   begin
   print_string "positif";
   end
else
   begin
   print_string "négatif";
   end
;
```

```
File "test.ml", line 6, characters 0-4:
Syntax error
```

C'est-à-dire qu'il y a une erreur de syntaxe au niveau du `else`.

1.3.4 Exercices

★ Écrire un programme qui affiche `"égal à 4"` si le nombre fourni par l'utilisateur est égal à 4, et `"différent de 4"` sinon. Dans les deux cas, il faut ensuite afficher un retour à la ligne.

\- -

```
let x = read_int() in
if x = 4 then
   begin
   print_string "égal à 4";
   end
else
   begin
   print_string "différent de 4";
   end
;
print_newline();
```

Remarque : il n'est pas souhaitable de mettre un `print_newline()` dans chacun des deux blocs, vu qu'on peut se contenter d'un seul à la fin.

★ Écrire un programme qui demande deux nombres, et affiche le premier nombre, puis `"et"`, le second nombre, puis `"sont"`, et enfin `"égaux"` ou `"différents"` selon les cas. Attention à bien mettre des espaces là où il faut.

_ _

```
let x = read_int() in
let y = read_int() in
print_int x;
print_string " et ";
print_int y;
print_string " sont ";
if x = y then
   begin
   print_string "égaux";
   end
else
   begin
   print_string "différents";
   end
;
```

Ce qu'il ne fallait pas faire, c'est écrire du code en double :

```
let x = read_int() in
let y = read_int() in
if x = y then
   begin
   print_int x;
   print_string " et ";
   print_int y;
   print_string " sont ";
   print_string "égaux";
   end
else
   begin
   print_int x;
   print_string " et ";
   print_int y;
   print_string " sont ";
   print_string "différents";
   end
```

```
;
```

★ Écrire un programme qui demande deux nombres à l'utilisateur, et qui affiche la valeur du plus grand d'entre eux.

_ _

```
let x = read_int() in
let y = read_int() in
if x > y then
   begin
   print_int x;
   end
else
   begin
   print_int y;
   end
;
```

Remarque : on peut aussi faire le test `x >= y`, vu que lorsque `x = y`, n'importe lequel des deux blocs affiche un résultat correct.

★ Écrire un programme qui lit un texte, et affiche `"c'est moi"` si le texte est égal à votre prénom, et `"c'est quelqu'un d'autre"` sinon.

_ _

```
let prenom = read_line() in
if prenom = "arthur" then
   begin
   print_string "c'est moi";
   end
else
   begin
   print_string "c'est quelqu'un d'autre";
   end
;
```

★ Modifier le programme précédent pour qu'il n'affiche rien du tout s'il s'agit de quelqu'un d'autre.

_ _

Il suffit d'enlever la ligne avec le `print_string "c'est quelqu'un d'autre"`.

```
let prenom = read_line() in
if prenom = "arthur" then
   begin
   print_string "c'est moi";
   end
else
   begin
   end
;
```

1.3.5 Réductions des blocs

▶ Il s'agit d'alléger les notations dans les structures conditionnelles (avec `if`) lorsque des blocs sont formés d'une seule instruction. Voici la règle :

```
begin
instruction;
end
```

se simplifie simplement en `"instruction"`.

Par exemple :

```
let x = read_int() in
if x = 4
then
   begin
   print_string "égal à 4";
   end
else
   begin
   print_string "différent de 4";
   end
;
print_newline();
```

Se réduit ainsi (attention aux points-virgules qui ont été retirés) :

```
let x = read_int() in
if x = 4 then
   print_string "égal à 4"
else
   print_string "différent de 4"
;
print_newline();
```

▶ Regardez bien le code précédent : la structure `if then else ;` prend 5 lignes. En programmation, le but est d'obtenir un code tel qu'on puisse comprendre le plus rapidement possible la structure du programme.

Dans une structure `if`, lorsque les blocs du `then` et du `else` sont réduits à une instruction chacun, on adopte la présentation suivante :

```
if (...condition...)
   then (...instruction du then...)
   else (...instruction du else...);
```

Dans notre exemple :

```
let x = read_int() in
if x = 4
   then print_string "égal à 4"
   else print_string "différent de 4";
print_newline();
```

Intérêts :

- Comme avant, les tabulations avant le `then` et le `else` montrent que ces lignes ne sont pas forcément exécutés : leur exécution dépend de la condition.
- Comme avant, il y a une symétrie entre les divers cas possibles : soit l'instruction après le `then` est exécutée, soit c'est celle après le `else`.
- Maintenant, de plus, trois lignes suffisent.

Inconvénient : il faut mettre un point virgule à la fin de la ligne du `else`, pour terminer la structure `if`, alors qu'il ne faut pas en mettre après le `then`. Cela nuit un peu à la symétrie du code, mais on ne va tout de même pas gâcher une ligne entière juste pour mettre un point virgule.

1.3.6 Erreurs et exercice

Si vous mettez un point-virgule en trop à la fin du bloc du `then`, vous obtiendrez une erreur sur le `else` :

```
let x = read_int() in
if x = 4
   then print_string "égal à 4";
   else print_string "différent de 4";
```

```
File "test.ml", line 4, characters 3-7:
Syntax error
```

Si au contraire vous oubliez le point-virgule après le `else`, vous obtiendrez un message d'erreur sur la ligne suivante.

★ En utilisant la nouvelle présentation du `if`, écrivez un programme qui demande deux nombres, et affiche juste `"égaux"` ou `"différents"` selon les cas.

_ _

```
let x = read_int() in
let y = read_int() in
if x = y
    then print_string "égaux"
    else print_string "différents";
```

1.3.7 Blocs vides, `else` implicite

▶ Il s'agit cette fois d'alléger les notations dans les structures conditionnelles (avec `if`) lorsque les blocs n'ont aucune instruction. Voici la première règle :

Un bloc vide `begin end` peut s'écrire aussi comme un couple de parenthèses : `()`.

Exemple sur une variante d'un exercice précédent :

```
let prenom = read_line() in
if prenom = "arthur" then
   begin
   print_string "c'est moi";
   print_newline();
```

```
    end
else
    begin
    end
;
```

Se simplifie en :

```
let prenom = read_line() in
if prenom = "arthur" then
    begin
    print_string "c'est moi";
    print_newline();
    end
else
    ()
;
```

▶ Et maintenant une seconde règle qui permet de simplifier encore plus :

Dans une structure `if`, on peut se passer du `"else ()"`.

Illustration :

```
let prenom = read_line() in
if prenom = "arthur" then
    begin
    print_string "c'est moi";
    print_newline();
    end
;
```

Notez que le point-virgule correspondant à la fin du bloc `if` se retrouve maintenant à la suite du bloc du `then`.

Dans un tel cas, on peut si on a envie placer le point-virgule après le `end` :

```
let prenom = read_line() in
if prenom = "arthur" then
    begin
    print_string "c'est moi";
    print_newline();
    end;
```

C'est un peu une question de goût.

★ Écrire un programme qui lit un entier, et affiche `"positif"` si celui si est positif ou nul, et rien sinon.

– –

```
let x = read_int() in
if x >= 0
    then print_string "positif";
```

★ Exercice : comment pourriez-vous simplifier le code suivant ? Indication : pensez à utiliser le test de différence, qui s'écrit `<>`.

```
let prenom = read_line() in
if prenom = "arthur" then
   ()
else
   begin
   print_string "Halte là !";
   print_newline();
   end
;
```

_ _

La solution dans ce cas est de tester la condition contraire :

```
let prenom = read_line() in
if prenom <> "arthur" then
   begin
   print_string "Halte là !";
   print_newline();
   end
else
   ()
;
```

Ce qui permet alors d'effectuer la simplification du `else`.

```
let prenom = read_line() in
if prenom <> "arthur" then
   begin
   print_string "Halte là !";
   print_newline();
   end;
```

1.3.8 Le `else if`

★ Écrire un programme qui demande un nombre à l'utilisateur, et affiche `"c'est 4"`, ou bien `"c'est 8"`, ou alors `"autre valeur"`, selon le cas. Indication : on peut réaliser un `if` à l'intérieur d'un bloc d'un autre `if`.

_ _

```
let x = read_int() in
if x = 4 then
   print_string "c'est 4"
else
   begin
   if x = 8
      then print_string "c'est 8"
      else print_string "autre valeur";
   end
;
```

Remarque : la solution n'est pas unique, et on peut très bien aboutir à un résultat correct avec des tests différents.

▶ On va pouvoir simplifier le code précédent grâce à la règle suivante :

Lorsque le bloc du `else` contient lui-même une structure `if`, on peut enlever les délimiteurs du bloc `begin` et `end`. Dans ce cas, il faut aussi enlever le point-virgule terminant le bloc imbriqué.

Enlevons donc le `begin` et le `end`, ainsi que le point-virgule après le `print_string "autre valeur"` :

```
let x = read_int() in
if x = 4 then
   print_string "égal 4"
else
   if x = 8
      then print_string "égal 8"
      else print_string "autre valeur"
;
```

Dans un tel cas, afin de mettre en valeur la symétrie entre les trois cas possibles, on présentera le code de la manière suivante :

```
let x = read_int() in
if x = 4 then
   print_string "égal 4"
else if x = 8 then
   print_string "égal 8"
else
   print_string "autre valeur"
;
```

On appelle cela une structure `if` avec des `else if`. Généralisons ce principe.

▶ Voici une structure avec des `else if` :

```
if (...1ere condition...) then
   (...bloc du 1er then...)
else if (...2ème condition...) then
   (...bloc du 2eme then...)
else if (...3ème condition...) then
   (...bloc du 3eme then...)
...
...
else
   ( ...bloc du else...)
;
```

Remarques : on peut mettre autant de `else if` que l'on souhaite, le dernier `else` pouvant être enlevé si le bloc associé est vide, et la réduction des blocs contenant une seule instruction fonctionne toujours.

Fonctionnement de la structure `if` avec des `else if` :

- On effectue le test après le premier `if`.
- Si celui-ci se révèle vrai, on exécute le bloc du `then` juste après, et on continue après le point-virgule terminant la structure

- Dans le cas contraire, on va jusqu'au `else if` qui suit et on recommence à appliquer cette même règle.
- Si aucune des conditions après les `if` n'a été réalisée, alors on effectue le bloc du `else`. Ce bloc peut être implicite, et dans un tel cas on ne fait rien.

★ Ecrivez un programme qui affiche `"c'est moi"` lorsque vous tapez votre prénom en minuscule, et `"C'EST MOI"` lorsque vous le tapez en majuscule, et rien dans les autres cas.

_ _

```
let prenom = read_line() in
if prenom = "arthur" then
   print_string "c'est moi"
else if prenom = "ARTHUR" then
   print_string "C'EST MOI"
;
```

1.4 Répétitions

1.4.1 Répéter un bloc

► Un des avantages des programmes sur les humains est qu'ils sont capables d'effectuer un traitement simple un très grand nombre de fois, rapidement et sans fatiguer. Pour en profiter, il nous faut une structure permettant de répéter un morceau de code un nombre de fois déterminé. Nous allons voir une première méthode pour ce faire, il s'agit de la boucle `for`. Voici la bête :

```
for i = 1 to (...nombre de répétition...) do
   (...corps de la boucle for...)
done;
```

Le corps de la boucle consiste en une succession d'instructions quelconques.

Fonctionnement de la boucle `for`, utilisée pour répéter un bloc :

1. La première ligne permet de préciser le nombre de fois qu'il faut répéter la boucle.
2. Les bornes `do` et `done` délimitent le bloc qu'on va répéter.
3. Le corps de la boucle `for` sera copié autant de fois qu'indiqué à la première ligne.
4. L'ensemble de la structure, formant une grosse instruction, doit être terminé par un point-virgule.

Remarque : la syntaxe de la première ligne peut paraître bien compliqué pour ce que l'on veut faire, mais c'est dû au fait qu'il s'agit d'une syntaxe plus générale, comme on le verra peu après.

Par exemple le code suivant :

```
print_string "bonjour";
print_newline();
print_string "bonjour";
print_newline();
print_string "bonjour";
print_newline();
```

D'après la règle énoncée plus haut, ce code peut se condenser en :

```
for i = 1 to 3 do
   print_string "bonjour";
   print_newline();
done;
```

★ Ecrivez un programme qui demande à l'utilisateur le nombre de fois que votre programme doit afficher `"bonjour"`, suivi à chaque fois d'un retour à la ligne.

_ _

```
let repete = read_int() in
for i = 1 to repete do
   print_string "bonjour";
   print_newline();
done;
```

► Les déclarations réalisées à l'intérieur du corps de la boucle ont une portée limitée à ce corps.

Par exemple :

```
for i = 1 to 3 do
   let x = 8 in
   print_int x;
done;
```

est en quelques sortes équivalent à :

```
begin
   let x = 8 in
   print_int x;
end;
begin
   let x = 8 in
   print_int x;
end;
begin
   let x = 8 in
   print_int x;
end;
```

A chaque `end`, l'association du `x` à la valeur 8 se termine, et c'est un autre bloc qui commence. A la suite de tout ce code, `x` n'est associé à rien.

1.4.2 Itérer un bloc

► Répéter exactement la même chose ne sert en fait pas très souvent. On a plutôt besoin de répéter presque la même chose, mais pour un paramètre différent à chaque fois. En voici un exemple.

```
print_int 5;
print_newline();
print_int 6;
```

```
print_newline();
print_int 7;
```

On a 3 fois le même code, avec juste un entier qui change. Réécrivons cela en trois blocs, en montrant bien qu'il n'y a qu'un entier qui change, et que c'est toujours au même endroit.

```
begin
   let i = 5 in
   print_int i;
   print_newline();
end;
begin
   let i = 6 in
   print_int i;
   print_newline();
end;
begin
   let i = 7 in
   print_int i;
   print_newline();
end;
```

La boucle `for`, dans son cadre général, permet exactement de condenser ce type de code. Voici la structure, avec les traductions suivantes : `for = pour`, `to = jusqu'à`, `do = faire`, `done = fini`.

```
for i = (...i la première fois...) to (...i la dernière fois...) do
   (...corps de la boucle for...)
done;
```

Vocabulaire : la lettre `i` s'appelle le compteur. La valeur de ce compteur lors de la première fois est la `"valeur initiale"`, et la dernière fois, c'est la `"valeur finale"`. On dit que l'on effectue une `"itération"` sur les valeurs de `i`.

Résumons :

```
for compteur = valeur_initiale to valeur_finale do
   (...corps de la boucle for...)
done;
```

Pour chaque valeur entière du compteur `i` comprise entre les deux bornes données (incluses) et dans l'ordre croissant, copier le bloc du for, et y déclarer le nom `i` comme étant associé à cette valeur.

Fonctionnement de la boucle `for` utilisé pour itérer un bloc avec un compteur :

1. Évaluer la valeur initiale et la valeur finale du compteur, ce sont les bornes.
2. Prendre dans l'ordre croissant tous les entiers compris entre ces bornes, incluses.
3. Pour chaque entier, copier le corps du for, et y déclarer le compteur égal à cet entier.

Remarque : les boucles for fonctionnent uniquement pour des entiers relatifs (c'est-à-dire positifs ou négatifs), et rien d'autre. N'essayez pas de faire prendre des valeurs réelles ou des lettres au compteur, ça ne peut pas marcher.

On peut condenser le code d'avant :

```
for i = 5 to 7 do
   print_int i;
   print_newline();
done;
```

★ Écrire un programme qui affiche pour chaque entier entre 1 et 20 : sa valeur, le texte `"au carré vaut"`, la valeur de son carré, et un retour à la ligne.

_ _

```
for i = 1 to 20 do
   print_int i;
   print_string " au carré vaut ";
   print_int (i * i);
   print_newline();
done;
```

► Dans toutes les boucles for, on a utilisé le nom `i` comme nom du compteur. On pourrait utiliser n'importe quel autre nom. Et il même fortement recommandé d'utiliser un autre nom s'il en existe un plus adapté. Nous verrons des exemples dans les exercices qui suivent.

1.4.3 A propos des bornes

★ Ecrivez un programme qui demande à l'utilisateur deux nombres, et qui les utilise comme valeur initiale et valeur finale du compteur d'une boucle `for`. Le corps de cette boucle consiste à afficher la valeur du compteur puis un espace.

Par exemple si les deux nombres sont 5 et 12, il faudra afficher :

```
5 6 7 8 9 10 11 12
```

_ _

```
let debut = read_int() in
let fin = read_int() in
for entier = debut to fin do
   print_int entier;
   print_string " ";
done;
```

Questions. Que se passe-t-il lorsque les deux nombres sont égaux ? Et lorsque le premier est supérieur au second ? Faîtes le test.

Lorsque la valeur initiale du compteur est égale à la valeur finale, la boucle n'est exécutée qu'une seule fois pour cette valeur. Lorsque la valeur initiale est supérieure strictement à la valeur finale, la boucle n'est exécutée aucune fois.

1.4.4 Descente

★ En utilisant une boucle `for`, écrivez un programme qui affiche tous les entiers de 20 à 1 inclus, dans l'ordre décroissant.

– –

Il y a 20 valeurs à afficher. Utilisons donc une boucle `for` avec un compteur `i` variant de 1 à 20, pour pouvoir itérer 20 fois. Comme les valeurs de `i` sont croissantes et que l'on veut des valeurs décroissantes, il va falloir utiliser `-i`. En effet, l'opposé de `i` est décroissant lorsque `i` est croissant : `-i` varie ainsi de `-1` à `-20`.

Ce n'est pas tout à fait ce que l'on veut. Lorsque `-i = -1`, on veut afficher 20, puis lorsque `-i = -2`, on veut afficher 19, ..., et à la fin lorsque `-i = -20`, on veut afficher 1. On voit la formule apparaître : la valeur à afficher est `-i + 21`. Voici donc le code :

```
for i = 1 to 20 do
   print_int (21 - i);
   print_newline();
done;
```

Variante possible :

```
for i = 0 to 19 do
   print_int (20 - i);
   print_newline();
done;
```

► Il existe une variante de la boucle for qui permet de faire la même chose plus facilement. Pour cela, il suffit d'utiliser `downto` à la place de `to`. Traduction : `downto = en descendant jusqu'à`. Un petit exemple :

```
for i = 20 downto 1 do
   print_int i;
   print_newline();
done;
```

Remplacer `to` par `downto` dans une boucle `for` pour avoir des valeurs du compteur décroissantes.

1.4.5 Erreurs

► Comme d'habitude, prenez le temps de regarder sur un exemple les messages d'erreurs de construction de la structure. Ce n'est vraiment pas du temps perdu.

Simulons ainsi l'oubli du `do` :

```
for i = 5 to 7
   print_int i;
   print_newline();
done;
```

```
File "test.ml", line 4, characters 0-4:
Syntax error
```

C'est une erreur de syntaxe au niveau du `done`, un peu comme lorsqu'on oublie un `then` dans une structure `if`, et qu'on obtient une erreur sur le `else`.

1.4.6 Exercices

★ Écrire un programme qui demande un mot, appelé `motif`, et un nombre nommé `nb`, et affiche `nb` fois le mot `motif`, en séparant ces mots par des espaces.

_ _

```
let motif = read_line() in
let nb = read_int() in
for i = 1 to nb do
   print_string motif;
   print_string " ";
done;
```

★ Écrire un programme qui demande un entier à l'utilisateur, et qui affiche 10 entiers consécutifs en montant à partir de celui-ci, et en les séparant par des espaces.

_ _

Supposons que l'utilisateur donne l'entier 5. Pour afficher 10 nombres consécutifs à partir de 5, il va falloir afficher tous les nombres entre 5 et 14. Il faut faire attention : il y a 11 nombres entre 5 et 15 inclus. Ainsi on veut afficher tous les nombres entre `5` et `5 + 9`. Ce raisonnement se généralise à n'importe quel nombre de départ, et on obtient le code :

```
let depart = read_int() in
for i = depart to depart + 9 do
   print_int i;
   print_string " ";
done;
```

1.5 Exercices

1.5.1 Signe d'un entier

★ Ecrivez un programme qui demande à l'utilisateur un entier relatif, et qui en affiche la valeur absolue.

_ _

```
let x = read_int() in
if x >= 0
   then print_int x
   else print_int (- x);
```

★ Ecrivez maintenant un programme qui affiche `"positif"`, `"négatif"`, ou `"nul"` en fonction du signe d'un nombre donné.

_ _

Il y a beaucoup de façon de faire. Il suffit en effet de réaliser deux tests, et si aucun des deux n'est réalisé, c'est qu'on est dans le troisième cas. Par exemple :

```
let x = read_int() in
if x > 0 then
   print_string "positif"
else if x = 0 then
   print_string "nul"
else
   print_string "négatif";
```

1.5.2 Minimum de trois nombres

★ Ecrivez un programme qui demande trois nombres à l'utilisateur et qui affiche le texte `"Le plus petit est :"`, suivi de la valeur du plus petit des trois nombres. Vous devez utiliser des structures `if` afin de déterminer cette valeur.

_ _

On commence par comparer les deux premiers nombres `x` et `y`. Une fois qu'on connaît le plus petit des deux, il n'y a plus qu'à le comparer avec le troisième nombre `z`. Traduit en code :

```
let x = read_int() in
let y = read_int() in
let z = read_int() in
print_string "Le plus petit est : ";
if x < y
then
   begin
   if x < z
      then print_int x
      else print_int z;
   end
else
   begin
   if y < z
      then print_int y
      else print_int z;
   end
;
```

On pourra écrire une solution beaucoup plus élégante lorsqu'on connaîtra les fonctions (chapitre 3).

1.5.3 Compte à rebours

★ Ecrivez un programme qui réalise un compte à rebours avant de se terminer. Il doit afficher `"Fin du programme dans i lignes"`, pour `i` partant d'une limite `depart` renseignée par l'utilisateur et descendant jusqu'à un (avec `depart` supérieur ou égal à 1). A la fin, affichez le texte `"Le programme est terminé"`. Par exemple, si l'utilisateur donne l'entier 3, il faut afficher :

```
Fin du programme dans 3 lignes
Fin du programme dans 2 lignes
Fin du programme dans 1 lignes
Le programme est terminé
```

```
let depart = read_int() in
for i = depart downto 1 do
   print_string "Fin du programme dans ";
   print_int i;
   print_string " lignes";
   print_newline();
done;
print_string "Le programme est terminé";
```

★ Dans le problème précédent, l'affichage contient une faute de grammaire lorsque le compteur `i` prend la valeur 1 puisqu'il faudrait afficher `"Fin du programme dans 1 ligne"`. Ecrivez donc une variante du programme précédent afin de corriger cette erreur.

Une première solution consiste à arrêter la boucle à `i = 2`, et de traiter le cas `i = 1` à part :

```
let depart = read_int() in
for i = depart downto 2 do
   print_string "Fin du programme dans ";
   print_int i;
   print_string " lignes";
   print_newline();
done;
print_string "Fin du programme dans 1 ligne";
print_newline();
print_string "Le programme est terminé";
```

Question : pourquoi le code précédent fonctionne-t-il même lorsque `depart = 1` ?

Réponse : comme la valeur de départ du compteur (`depart`) est inférieure à 2, le corps de la boucle n'est pas du tout exécuté. Il reste donc simplement les trois dernières lignes après la boucle.

★ Une seconde solution consiste à faire un test à l'intérieur de la boucle pour savoir s'il faut écrire `"lignes"` ou `"ligne"`. Si vous ne l'avez pas déjà fait, essayez de coder cette solution.

```
let depart = read_int() in
for i = depart downto 1 do
   print_string "Fin du programme dans ";
   print_int i;
   if i = 1
      then print_string " ligne"
      else print_string " lignes";
   print_newline();
done;
print_string "Le programme est terminé";
```

★ Une troisième manière de coder le programme : écrire `"ligne"`, et rajouter ensuite le `"s"` si nécessaire. Faîtes la petite modification du code pour essayer cette autre technique.

```
let depart = read_int() in
for i = depart downto 1 do
   print_string "Fin du programme dans ";
   print_int i;
   print_string " ligne";
   if i > 1
      then print_string "s";
   print_newline();
done;
print_string "Le programme est terminé";
```

On voit bien que tout serait plus simple si la grammaire n'était pas aussi subtile.

1.5.4 Lignes, rectangles et triangles

▶ Dans cette partie, on va dessiner les objets suivants :

```
Ligne : XXXXXXXX   Rectangle : XXXXXX   Triangle : X
                               XXXXXX               XX
                               XXXXXX               XXX
                               XXXXXX               XXXX
```

★ Ecrivez un programme qui demande un nombre `nb_colonnes` à l'utilisateur, puis qui affiche `nb_colonnes` fois de suite la lettre `"X"`. Toutes les lettres doivent être placées sur la même ligne, sans espaces entre elles, et il faut réaliser un retour à la ligne à la fin. Remarque : `nb_` est une abréviation courante pour `"nombre de"`.

_ _

C'est une simple répétition de l'instruction `print_string "X"` :

```
let nb_colonnes = read_int() in
for colonne = 1 to nb_colonnes do
   print_string "X";
done;
print_newline();
```

On a utilisé `colonne` comme nom de compteur, car c'est ce qu'il représente. On affiche en effet un `"X"` sur chaque colonne, de la première jusqu'à la dernière.

Vous remarquerez que le programme fonctionne même pour `nb_colonnes = 0`, puisque rien n'est affiché.

★ A l'aide de ce code, écrivez maintenant un programme qui demande à l'utilisateur deux nombres `nb_lignes` puis `nb_colonnes`, et qui affiche `nb_lignes` lignes contenant chacune `nb_colonnes` fois le caractère `"X"`. On dessinera ainsi un joli rectangle.

_ _

On place le code précédent à l'intérieur d'une boucle for qui sera répétée `nb_lignes` fois, sauf la déclaration de `nb_colonnes` qui se met au début, juste après celle de `nb_lignes`.

```
let nb_lignes = read_int() in
let nb_colonnes = read_int() in
for ligne = 1 to nb_lignes do
   for colonne = 1 to nb_colonnes do
      print_string "X";
   done;
   print_newline();
done;
```

Pour les lignes, on a utilisé le compteur `ligne`, naturellement.

★ On veut maintenant écrire un programme qui dessine un triangle rempli de `"X"`. On demande pour cela à l'utilisateur un entier `taille`. Après on rempli le triangle : sur la première ligne on met un `"X"`, sur la deuxième on en met deux, sur la troisième trois, etc... jusqu'à la ligne numéro `taille`, sur la quelle on placera `taille` fois la lettre `"X"`.

– –

Allons-y étape par étape. Sur la ligne numéro `ligne`, on veut afficher `ligne` fois le caractère `"X"`. Ca, on sait le faire :

```
for colonne = 1 to ligne do
   print_string "X";
done;
print_newline();
```

Maintenant, on veut faire cela pour toutes les lignes, de la première, où `ligne = 1`, à la dernière, où `ligne = taille`. On met donc le code précédent dans une boucle for, avec le compteur `ligne` variant de `1` à `taille`.

```
let taille = read_int() in
for ligne = 1 to taille do
   for colonne = 1 to ligne do
      print_string "X";
   done;
   print_newline()
done;
```

1.5.5 Table de multiplication

★ Le but est d'afficher une grande table de multiplication pour tous les entiers jusqu'à 10. Par exemple, à la ligne 4 sur la colonne 5 dans la table, on souhait lire le produit de 4 par 5, soit 20. Voici à quoi ressemble la table :

```
1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30
4       8       12      16      20      24      28      32      36      40
5       10      15      20      25      30      35      40      45      50
6       12      18      24      30      36      42      48      54      60
7       14      21      28      35      42      49      56      63      70
8       16      24      32      40      48      56      64      72      80
9       18      27      36      45      54      63      72      81      90
10      20      30      40      50      60      70      80      90      100
```

Plus généralement, sur la ligne `i`, à la colonne j, on doit trouver la valeur du produit des entiers `i` et `j`.

Pour que les nombres soient bien alignés sur des colonnes, il vous faudra utiliser des tabulations. Pour cela, effectuez `print_string "\t"` entre chaque valeur que vous affichez sur une même ligne. On verra d'où vient ce caractère `'\t'` par la suite.

_ _

On s'inspire du code qui nous a permis d'afficher le carré rempli de `"X"`. A chaque fois que l'on affichait un `"X"`, il va maintenant falloir afficher le produit des compteurs `ligne` et `colonne`, suivi d'une tabulation.

```
for ligne = 1 to 10 do
   for colonne = 1 to 10 do
      print_int (ligne * colonne);
      print_string "\t";
   done;
   print_newline();
done;
```

Et voilà !

Chapitre 2

float, int, ref

2.1 Réels

2.1.1 Opérations

▶ Pour l'instant on n'a utilisé que des nombres entiers. On a vu comment les lire avec `read_int()`, les afficher avec `print_int()`, et faire des opérations de bases dessus : addition `+`, soustraction `-`, et multiplication `*`. Notez que la division de deux entiers n'étant pas toujours un entier, le symbole `/` prendra un sens différent de la division usuelle, comme on l'expliquera dans la prochaine section.

Pour désigner la façon dont un ordinateur manipule les nombres réels, on utilise l'expression de `"nombres flottants"`, `"float"` en anglais. En français les réels ont des virgules, mais en anglais il y a un point, et ainsi `4.56` vaut quatre virgule cinquante-six. Même si le nombre n'a pas de chiffres après la virgule, pour qu'il soit considéré comme un float, il faut le faire suivre d'un point. Ainsi `4.` est un réel, alors que `4` est toujours un entier. Pour les nombres négatifs, il suffit de mettre un moins devant : `-234.81` par exemple. Enfin il est possible d'utiliser la notation scientifique : `1.27e12` signifie $1.27 * 10^{12}$.

A chaque opération sur les entiers, il va correspondre l'analogue pour les floats. On les lira avec `read_float()`, de la même manière qu'avec `read_int()`. La fonction `read_float()` est d'ailleurs capable de lire un réel même s'il est donné par l'utilisateur sous la forme d'un entier, comme par exemple `4` sans point à la fin. Pour afficher des float, on utilisera `print_float` suivi de la valeur du réel.

Pour les opérations, il faut faire très attention car les symboles sont tous suivis d'un point. Cela traduit différentes méthodes de calculs entres les entiers et les réels. Par exemple, une addition de deux réels n'est pas réalisée par un ordinateur de la même manière que l'addition de deux entiers. On a donc l'addition `+.` la soustraction `-.` la multiplication `*.` et enfin la division `/.` qui forment les quatre opérations.

Il est impossible de mélanger des int et des float sans soins particuliers.

▶ Voici un programme qui demande à l'utilisateur un réel, qui affiche d'une part le produit de ce nombre par `3.4`, et d'autre part la valeur de l'opposé de ce nombre.

```
let x = read_float() in
print_float (3.4 *. x);
print_newline();
print_float (-. x);
```

Remarque : `print_float (-x)` ne marche pas, car le `-` permet d'opposer des entiers, et pas des réels.

2.1.2 Erreurs

▶ L'erreur fréquente d'oubli de parenthèses, qu'on a déjà évoqué pour les entiers :

```
print_float -5.448;
```

```
File "test.ml", line 1, characters 0-11:
This expression has type float -> unit but is here used with type int
```

L'erreur se situe sur le `print_float`, qui est normalement de type `"float -> unit"` (c'est une fonction, on le verra). Le compilateur dit qu'on devrait normalement avoir un entier à cet endroit. L'explication est qu'il essaie d'effectuer une soustraction de deux entiers, avec le signe moins, comme cela :

```
(print_float) - (5.448);
```

▶ Si vous tentez de mélanger des valeurs ou des opérations réelles avec des valeurs ou des opérations entières, une erreur de type se produira. Un petit exemple :

```
let a = read_float() in
print_float (3 +. a);
```

Donne une erreur sur le `3` :

```
File "test.ml", line 2, characters 13-14:
This expression has type int but is here used with type float
```

Ce message est facile à comprendre : le 3 qu'il désigne est un int (entier), alors qu'il devrait être un float (réel), puisqu'on a utilisé l'opérateur `+.` qui additionne deux floats.

▶ C'est bien connu, il ne faut jamais diviser par zéro. Tout simplement parce qu'on ne peut pas prévoir le résultat.

Lorsqu'on effectue une division par zéro en calculant sur des réels, un code spécial est utilisé pour désigner le résultat. Sous Windows :

- `1.#INF` et `-1.#INF` respectivement pour plus et moins l'infini.
- `1.#IND` et `-1.#IND` pour des valeurs indéterminées.

Sous linux :

- `inf.` et `-inf.` respectivement pour plus et moins l'infini.
- `nan.` et `-nan.` pour des valeurs indéterminées.

Voici un exemple de code faisant apparaître ces valeurs. Sous Windows :

```
print_float (5. /. 0.);
print_newline();
print_float (-1. /. 0.);
print_newline();
print_float (0. *. ( 5. /. 0.));
```

```
1.#INF
-1.#INF
-1.#IND
```

Et sous Linux :

```
inf.
-inf.
nan.
```

Ce n'est pas très important de comprendre le fonctionnement détaillé de ces subtilités. L'essentiel est de se souvenir que l'affichage de valeurs de ce type est dû à des problèmes de calculs qui ne sont pas définis.

2.1.3 Exercices

★ Écrire un programme qui demande deux réels et qui affiche le carré de leur somme.

_ _

Voilà ce qu'il ne faut pas faire :

```
let x = read_float() in
let y = read_float() in
print_float ( (x +. y) *. (x +. y) );
```

Le problème dans ce code est qu'on répète deux fois le même code, à savoir `(x +. y)`. Afin d'éviter cela, il faut mieux poser la valeur de cette somme avant, et d'en faire le produit après.

```
let x = read_float() in
let y = read_float() in
let s = x +. y in
print_float (s *. s);
```

Note : on va voir l'opérateur puissance juste après.

2.1.4 Valeur absolue, puissances

▶ Pour calculer la valeur absolue d'un réel `x`, on fait `abs_float x`. Exemple d'utilisation :

```
let x = read_float() in
print_float (abs_float x);
```

Si le paramètre de la fonction est le résultat d'un calcul, il faut le mettre entre parenthèses. Ainsi pour prendre la valeur absolue d'une différence :

```
let x = read_float() in
let y = read_float() in
let r = abs_float (x -. y) in
print_float r;
```

▶ Pour prendre la racine carrée d'un nombre réel `x`, on fait `sqrt x`. Mais attention, la racine carrée d'un nombre négatif n'étant pas définie, on risque une forme indéterminée. En fournissant le nombre `-2.3` par exemple au code suivant :

```
let x = read_float() in
print_float (sqrt x);
```

le programme affiche `-1.#IND`.

▶ Pour élever un nombre à une puissance, on utilise le symbole `**` (deux signes fois côte à côte), qui se place entre le nombre et sa puissance. Il faut faire attention au fait que ces deux valeurs sont des nombres réels tous les deux. Ainsi pour mettre un nombre `x` au cube, il faudra faire `x ** 3.0` :

```
let x = read_float() in
print_float (x ** 3.0);
```

Remarque : en fait, `sqrt x` est simplement équivalent à `x ** 0.5`.

★ Écrire un programme qui donne la norme d'un vecteur dont l'utilisateur donne les trois coordonnées x, y et z de son vecteur. On rappelle la formule de la norme : $\sqrt{x^2 + y^2 + z^2}$

– –

Première solution avec des produits :

```
let x = read_float() in
let y = read_float() in
let z = read_float() in
let norme = sqrt (x *. x +. y *. y +. z *. z) in
print_float norme;
```

Seconde solution avec des puissances :

```
let x = read_float() in
let y = read_float() in
let z = read_float() in
let norme = sqrt (x ** 2. +. y ** 2. +. z ** 2.) in
print_float norme;
```

2.1.5 Maximum

▶ La fonction `max` donne le plus grand de deux nombres réels. Il faut pour cela placer les deux nombres après `max`, séparés par des espaces :

```
print_float (max 4.5 8.6);
```

affiche donc `8.6`.

La fonction `max` ne fonctionne que pour deux valeurs.

► La fonction `min` donne le plus petit de deux nombres réels, et s'utilise comme la fonction `max`. Par exemple :

```
let x = read_float() in
let y = read_float() in
let r = min (x +. 5.) (y. -. 5.) in
print_float r;
```

★ Ecrivez un programme qui demande trois nombres réels, et qui affiche le plus petit d'entre eux, en utilisant la fonction `min` et non des structures `if`.

_ _

```
let x = read_float() in
let y = read_float() in
let z = read_float() in
print_float (min (min x y) z);
```

Remarque : si par hasard vous avez écrit `(min (min x y) (min x z))`, vous avez une solution qui marche mais qui est inutilement compliquée.

2.1.6 Erreurs

► Il faut faire attention à la priorité des opérateurs. Par exemple, si l'objectif du programme est de calculer la valeur absolue de la différence des nombres `2.3` et `3.4`, alors le code suivant :

```
print_float (abs_float 2.3 -. 3.4);
```

compile parfaitement, mais pourtant il ne fait pas ce que l'on veut. En effet, il affiche : `-1.1` et non pas `1.1`. L'explication est très simple : les fonctions s'appliquent avant les opérations. Le code précédent est donc équivalent à :

```
print_float ( (abs_float 2.3) -. 3.4 );
```

soit encore, puisque `2.3` est positif :

```
print_float ( 2.3 -. 3.4 );
```

► Cette exemple montre qu'il ne suffit pas de tester une seule fois un programme pour se convaincre qu'il est correct. Supposons que le sujet d'un exercice soit de calculer la valeur absolue de la différence de deux nombres donnés, et qu'un programmeur écrive :

```
let x = read_float() in
let y = read_float() in
print_float (abs_float x -. y);
```

Ce programmeur peut alors tester son code en donnant les réels `5.6` et `-4.3`, il obtient la différence `9.9` et peut croire que son programme est correct. Pourtant, en testant avec `-4.3` puis `5.6`, il verrait que son programme est faux : il affiche `-1.3` et non `9.9`.

Conclusion : testez autant de situations différentes que possible pour vérifier un programme.

2.1.7 Autres fonctions

▶ Certaines fonctions mathématiques sont présentées ici. Si vous ne les connaissez pas, vous n'avez pas besoin de vous en soucier. Si vous pensez que vous en aurez besoin un jour, il n'est pas nécessaire de les apprendre par cœur tout de suite : vous pourrez toujours en retrouver la liste dans le manuel de référence du langage.

▶ Les fonctions exponentielles :

- `exp` pour l'exponentielle,
- `log` pour le logarithme naturel,
- `log10` pour le logarithme en base 10.

Par exemple, `exp (b *. (log a))` calcule `a` élevé à la puissance `b`, c'est-à-dire la même chose que `a ** b`.

▶ Les fonctions trigonométriques et trigonométriques hyperboliques, avec leurs réciproques. On les donne pour le cosinus, il y a les mêmes pour le sinus (sin) et la tangente (tan).

- `cos` pour le cosinus,
- `acos` pour la réciproque du cosinus : (`"arccos"` en français),
- `cosh` pour le cosinus hyperbolique (`"ch"` en français),
- `acosh` pour la réciproque du cosinus hyperbolique (`"argch"` en français),

Par exemple, `(acos (-1.))` et code(2. *. (asin 1.)) donnent tous deux la valeur de pi.

2.2 Entiers

2.2.1 Division entière et modulo

▶ L'opérateur `/` représente la division entière. C'est-à-dire qu'il calcul le quotient de deux entiers. L'opérateur `mod` calcule le reste de la division entière. Détaillons cela.

Supposons qu'un fermier dispose de `n` œufs, et qu'il souhaite remplir des boites de 12. L'expression `(n / 12)` donne le nombre de boites qu'il remplira entièrement. L'expression `(n mod 12)` donne le nombre d'œufs qu'il trouvera dans la boite incomplète. En particulier, lorsque `n` est un multiple de 12, il n'y a pas de boite incomplète, et dans ce cas `(n mod 12)` vaut 0.

Remarque : les divisions par zéro provoquent ici des erreurs. `(n / 0)` tout comme `(n mod 0)` amène l'erreur d'exécution suivante :

```
Fatal error: exception Division_by_zero;
```

★ Écrire un programme qui demande un entier à l'utilisateur et affiche *pair* ou *impair* selon les cas.

_ _

Il s'agit de tester si `n` est un multiple de 2, c'est-à-dire si `(n mod 2)` vaut 0.

```
let n = read_int() in
if n mod 2 = 0
   then print_string "pair"
   else print_string "impair";
```

On aurait aussi pu tester la condition inverse, à savoir si `n` est impair. On testerait alors si le reste de la division entière par 2 est 1.

```
let n = read_int() in
if n mod 2 = 1
   then print_string "impair"
   else print_string "pair";
```

2.2.2 Fonctions prédéfinies

► Comme pour les réels, on dispose de fonctions prédéfinies pour les entiers. Leur mode de fonctionnement est très similaire à celui des fonctions réelles.

- La valeur absolue d'un entier `n` s'écrit `abs n`.
- Les fonctions `min` et `max` fonctionnent pour les entiers aussi bien que pour les réels. Si `a` et `b` sont deux entiers `(min a b)` et `(max a b)` représentent respectivement leur minimum et leur maximum.

Remarque : `min`, `max`, et les opérateurs de comparaisons (`<, >, <=, >=, =, <>`) sont les seuls opérateurs qui fonctionnent à la fois sur des entiers et les réels. Il s'agit d'une exception, qui permet au programmeur d'écrire des tests sans alourdir les notations.

► Il existe deux fonctions si triviales qu'à priori elles ne servent à rien, mais qui nous permettront dans la suite d'écrire du code plus lisible, donc il faut les retenir.

La première fonction permet d'obtenir le nombre qui suit immédiatement un entier. Elle est nommée `succ` en abréviation de successeur. Ainsi pour un entier `n`, l'expression `succ n` est équivalente à `n+1`.

La seconde fonction permet d'obtenir le nombre précédent un entier, et se nomme `pred`. Ainsi l'expression `pred n` est équivalente à `n-1`.

2.2.3 Conversions

► Lorsqu'on dispose d'une valeur entière par exemple 2, et que l'on veut convertir cette valeur en une valeur réelle, c'est-à-dire `2.0`, il faut utiliser la fonction `float_of_int`, traduire `"réel à partir d'un entier"`.

La fonction `float_of_int` convertit un entier en le réel correspondant.

Par exemple :

```
let x = 2 in
print_int x;
print_newline();
```

```
let y = float_of_int x in
print_float y;
```

★ Ecrivez une fonction qui demande deux entiers à l'utilisateur, et qui renvoie la valeur réelle de la division exacte du premier par le second.

_ _

```
let x = read_int() in
let y = read_int() in
let divise = (float_of_int x) /. (float_of_int y) in
print_float divise;
```

★ Ecrivez une fonction qui affiche les valeurs des racines carrées des nombres entiers compris entre 0 et `k`, où `k` est donné par l'utilisateur. Affichez une valeur par ligne.

_ _

```
let k = read_int() in
for i = 0 to k do
   let racine = sqrt (float_of_int i) in
   print_float racine;
   print_newline();
done;
```

► Pour convertir un réel en un entier, c'est un peu plus délicat, car le réel ne tombe par forcément pile sur un entier. On dispose de la fonction `int_of_float`, traduction : `"entier à partir d'un réel"`, qui se contente d'effacer tous les chiffres après la virgule du nombre réel. Ainsi :

```
int_of_float 4.2 = 4
int_of_float 4.9 = 4
int_of_float (-4.2) = -4
int_of_float (-4.9) = -4
```

Exemple de programme :

```
let y = 4.2 in
let x = int_of_float y in
print_int x;
```

affiche 4.

Attention : la fonction `int_of_float` n'est pas la fonction mathématique partie entière. Cela à cause de sa façon de traiter les nombres négatifs. Par exemple la partie entière de `-4.2` est `-5` et non pas `-4`. Si vous avez besoin de calculer la vraie partie entière d'un réelle `x`, faîtes : `(int_of_float (floor x))`. En bref, `floor x` représente le plus grand réel inférieur à `x` qui n'ait que des zéros après la virgule.

2.2.4 Limitation des `int`

► Les nombres entiers ne peuvent pas être aussi grand ou aussi petit que l'on veut.

Un `int` est une valeur comprise entre `-1073741824` et `1073741823`.

On expliquera à la fin du cours pourquoi les limites se situent à ces valeurs précisément (les curieux remarqueront déjà qu'il s'agit d'une puissance de deux, à une unité près). Pour l'instant, le but est surtout d'être informé qu'il y a une limite, et surtout de savoir qu'aucun message d'erreur ne sera affiché si vous dépassez ces limites !

Mais quel nombre entier est le suivant de `1073741823` ? Écrivons un programme pour le voir :

```
print_int (1073741823 + 1);
```

L'affichage qu'on obtient est le nombre `-1073741824` ! Évidement, cela peut surprendre la première fois. C'est un peu comme si les nombres de type `int` tournaient en rond. Lorsqu'on arrive tout en haut des positifs, on repart de tout en bas des négatifs. Selon la même logique, le nombre juste en dessous de `-1073741824` est `1073741823`, comme le montre le programme :

```
print_int (-1073741824 - 1);
```

Conclusion :

Lorsqu'on travaille avec des `int`, il faut faire attention à ne pas dépasser les limites, car aucun message d'erreur ne signale les dépassements.

Remarque : pour faire des calculs avec des nombres plus grands, il faudra soit utiliser des fonctions spéciales, soit se contenter d'une approximation avec un nombre de type `float`. Dans une grande majorité des programmes dont l'objectif n'est pas de faire des maths, ces limites sur les `int` sont suffisamment grandes pour ne poser aucun problème.

2.3 Références

2.3.1 Problématique

▶ On veut écrire un programme qui demande `n` nombres à l'utilisateur, et en affiche la somme. Lorsque `n` est petit, on sait comment faire : lire tous les nombres avec des `let`, calculer alors la somme, et enfin afficher la valeur de cette somme. Mais si `n` est trop grand, on ne peut pas procéder ainsi. La solution consiste à ajouter successivement les valeurs données à un `total`, et d'afficher simplement le total à la fin :

```
soit total = 0;
faire n fois :
   demander un entier à l'utilisateur;
   ajouter cet entier au total;
;
afficher total;
```

Pour l'instant on n'est pas capable de coder ça. Pourquoi ?

Il n'est pas possible de modifier la valeur de `total`. Si on fait `let total = 0 in` au début du programme, `total` sera toujours associé à 0 dans la suite, et donc `afficher total` ne pourra afficher que 0.

On avait dit qu'on ne souhaitait pas que la valeur associée à un nom change. Ce souhait est toujours d'actualité, et on va arriver à résoudre le problème sans transgresser à cette règle. Voici comment.

▶ On va simplement dire que `total` est le nom d'une boite. Du début, jusqu'à la fin. Ainsi le nom `total` reste toujours associé à la même valeur : cette valeur, c'est une boite. La subtilité, c'est qu'on va alors pouvoir changer le contenu de la boite. Le changer comme on veut, quand on veut, où on veut.

Réécrivons donc ce que l'on veut faire, en utilisant une boite, nommée `total`. Au début, on crée cette boite, et on met la valeur 0 dedans : 0 est le contenu initial de la boite. Ensuite, à chaque fois que l'utilisateur donne un nombre, on effectue l'opération suivante :

- lire le contenu de la boite `total`,
- faire la somme de cette valeur et de l'entier donné par l'utilisateur,
- mettre cette somme dans la boite, à la place de ce qu'il y avait avant.

Version pseudo-code :

```
soit une nouvelle boite nommée total, avec 0 pour contenu
faire n fois :
   demander un entier à l'utilisateur;
   mettre dans la boite total : son contenu courant + l'entier donné;
;
afficher le contenu de la boite total;
```

2.3.2 Manipulations

▶ Avant de spécifier la syntaxe permettant de manipuler les boites, nous allons faire deux remarques fondamentales.

1) Une boite est fabriquée pour contenir un certain type d'objets. Elle ne pourra en aucun cas contenir des objets d'un autre type. Ainsi, on distinguera les `"boites à entiers"` des `"boites à réels"`, et des `"boites à texte"`, qui sont des objets de natures distinctes (on dira plus tard de **types** distincts).

2) Une boite n'est jamais vide. Il faut toujours mettre un contenu dans une boite, même si ce contenu n'est pas intéressant. Ainsi, dès la création d'une boite, il faudra donner un contenu, dit `"contenu initial"`.

Résumons :

Pour créer une boite, il faut donner un contenu initial. A tout instant, la boite contiendra un et un seul objet du même type que ce contenu initial.

Cette règle est relativement artificielle. L'objectif est surtout d'empêcher le programmeur de faire des erreurs. L'empêcher de mettre n'importe quel type de contenu dans n'importe quel boite, ou encore l'empêcher de demander le contenu d'une boite qui serait vide. L'intérêt est que si le programmeur se trompe, il sera prévenu au moment de la compilation, et il ne découvrira pas ce problèmes en utilisant son programme.

▶ D'abord un petit point vocabulaire : techniquement parlant, on utilise le mot `"référence"` plutôt que `"boite"`. De ce terme vient l'abréviation `ref`, qui va nous servir pour construire des références. Une boite conçue pour contenir des entiers sera donc appelée une `"référence sur un entier"`. Par ailleurs, lorsqu'on change le contenu d'une référence, on dit que l'on `"affecte"` la nouvelle valeur à cette référence.

▶ Voyons maintenant comment manipuler les références.

Pour créer une référence, on écrit le mot `ref`, suivi le contenu initial. Par exemple pour une référence contenant l'entier 15, on fait `ref 15`. Ensuite pour nommer cette référence, on utilise un let :

```
let b = ref 15 in
```

Pour obtenir le contenu d'une référence, on met un point d'exclamation devant son nom : `!nom`. Certaines personnes lisent le point d'exclamation : `"bang"`, et ainsi `!a` se lit `"bang a"`. Si on veut afficher le contenu de la boite précédente, on fait donc `print_int !b`. Ce qui donne au total :

```
let b = ref 15 in
print_int !b;
```

Remarque : il n'y a pas besoin de parenthèses autour de `!b` après le `print_int`, car l'opérateur `!` est prioritaire.

Pour affecter un nouveau contenu à une référence, on écrit son nom, puis le symbole `:=`, et enfin son nouveau contenu. Changeons ainsi le contenu de `b` en le faisant passer de 15 à 18, et affichons ensuite la valeur du nouveau contenu :

```
let b = ref 15 in
print_int !b;
b := 18;
print_int !b;
```

Il est extrêmement important de comprendre le principe des références, car ces principes s'étendront par la suite à d'autres structures fondamentales.

► Résumé des manipulations de références :

- `let nom = ref contenu in` pour construire une nouvelle référence.
- `!nom` pour accéder au contenu de la référence.
- `nom := nouveau_contenu` pour affecter un nouveau contenu à la référence.

2.3.3 Erreurs

► Le point d'exclamation sert à différencier le nom d'une boite de son contenu. Ainsi, `b` est une boite et `!b` est son contenu. L'oubli d'un point d'exclamation entraîne une erreur :

```
let b = ref 15 in
print_int b;
```

```
File "test.ml", line 2, characters 10-11:
This expression has type int ref but is here used with type int
```

Cette erreur est située sur le `b` après le `print_int`. Le compilateur nous dit que ce `b` a pour type `"int ref"`, traduire `"référence sur un entier"`. Mais on devrait avoir un `"int"`, un entier, puisque `print_int` affiche des entiers.

► L'oubli d'un `ref` :

```
let b = 15 in
b := 18;
print_int !b;
```

```
File "test.ml", line 2, characters 0-1:
This expression has type int but is here used with type 'a ref
```

L'erreur se situe sur le `b` de `b := 18`. Ce `b` est pour l'instant un entier, qu'on vient de poser égal à 15. Il est pourtant utilisé comme une `'a ref`. Qu'est-ce que cela veut dire ? Simplement `"référence sur quelque chose"`. Les `'a`, `'b` et autres lettres précédées d'un prime veulent dire `"un type indéterminé"`.

On peut se demander pourquoi le compilateur n'a pas vu qu'on essayait de mettre 18 dans `b`, et pourquoi il n'a donc pas dit qu'on utilisait ici `b` comme `int ref`. La raison est la suivante : il ne s'est même pas donné la peine d'aller lire jusqu'au 18. Au moment où il a vu `b :=` il signale déjà l'erreur, et s'arrête tout de suite après. En effet, l'opérateur d'affectation `:=` doit obligatoirement être précédé du nom d'une référence.

▶ Enfin l'écriture d'un simple signe `=` à la place d'un `:=` provoque une erreur de type :

```
let b = ref 15 in
b = 18;
print_int !b;
```

```
File "test.ml", line 2, characters 4-6:
This expression has type int but is here used with type int ref
```

Souvenez-vous : la seule fois qu'on a utilisé un signe égal en dehors d'un `let` c'était pour réaliser un test d'égalité dans une structure `if`, du genre `if x = 3 then ...`. Comme on le précisera plus tard, il est possible de réaliser des comparaisons aussi en dehors des structures `if`. C'est ce qui se passe ici : le compilateur pense que l'on essaie de tester si `b` et `18` sont égaux. Ce qui provoque un problème de type, vu que cela n'a pas de sens de regarder si une boite et un entier ont même valeur.

2.3.4 Exercices

★ Ecrivez un programme qui crée une boite nommée `b` contenant 10, puis qui augmente le contenu de cette boite de 5, et qui pour terminer affiche le contenu de la boite (qui sera 15). Contrainte : vous devez utiliser l'expression `!b + 5` pour calculer le nouveau contenu de la boite.

_ _

```
let b = ref 10 in
b := !b + 5;
print_int !b;
```

★ Ecrivez un programme qui crée une boite contenant un entier donné par l'utilisateur, et qui augmente ensuite le contenu de cette boite de 5, avant d'afficher ce contenu. N'oubliez pas de nommer la valeur donnée par l'utilisateur avant de faire le reste.

_ _

```
let initial = read_int() in
let b = ref initial in
b := !b + 5;
print_int !b;
```

Il est certes possible de faire : `let b = ref (read_int()) in`, mais c'est déconseillé, car on préfère toujours mettre une seule instruction par ligne. Donc d'abord on lit un entier, et ensuite on construit une boite contenant cet entier.

★ Ecrivez maintenant un programme qui demande un entier `n` à l'utilisateur, et qui lui demande ensuite `n` fois un entier,,pour afficher à la fin la somme de ces `n` entiers. Rappel du pseudo-code à traduire en Caml :

```
demander n à l'utilisateur
soit une nouvelle boite nommée total, avec 0 pour contenu
faire n fois :
   demander un entier à l'utilisateur;
   mettre dans la boite total : son contenu courant + l'entier donné;
;
afficher le contenu de la boite total;
```

_ _

```
let n = read_int() in
let total = ref 0 in
for i = 1 to n do
   let entier = read_int() in
   total := !total + entier;
done;
print_int !total;
```

★ Modifier le code précédent pour écrire un programme qui calcule la somme des entiers de 1 à `n`, où `n` est donné par l'utilisateur.

_ _

La modification à faire, c'est de remplacer le `read_int()` par la valeur du compteur `i`. On peut alors remplacer `entier` par `i` dans l'affectation. Voilà :

```
let n = read_int() in
let total = ref 0 in
for i = 1 to n do
   total := !total + i;
done;
print_int !total;
```

★ Modifier encore le code précédent pour écrire un programme qui calcule la somme des racines des nombres entiers de 1 à `n`, où `n` est donné par l'utilisateur. Vérifiez que pour `n = 100`, le résultat de votre programme soit bien `671.462947103`.

_ _

Cette fois si, il faut utiliser une boite contenant des réelles. Le contenu initial est alors `0.` (le zéro des réels). Pour prendre la racine de la valeur du compteur, il faut d'abord le convertir en un réel, avec `float_of_int` on le rappelle. Il ne faut pas oublier de mettre un `print_float` à la fin pour afficher le total.

```
let n = read_int() in
let total = ref 0. in
for i = 1 to n do
   let ajout = sqrt (float_of_int i) in
   total := !total +. ajout;
done;
print_float !total;
```

2.3.5 Incrémentation

★ Ecrivez un programme qui demande à l'utilisateur un entier `n`, puis qui lui demande `n` entiers, et affiche alors le nombre d'entiers fournis qui étaient supérieurs ou égaux à 5.

_ _

On utilise une boite nommée `quantite` qui retient le nombre d'entiers supérieurs à 5 qui ont déjà été fournis. A chaque fois que l'utilisateur donne un entier, on test s'il est supérieur à 5, et dans ce cas, on remplace augmente le contenu de quantité de 1.

```
let n = read_int() in
let quantite = ref 0 in
for i = 1 to n do
   let valeur = read_int() in
   if (valeur >= 5)
      then quantite := !quantite + 1;
done;
print_int !quantite;
```

▶ L'instruction `quantite := !quantite + 1` s'appelle une `"incrémentation"` de la référence `quantite`. Cette opération étant assez courante, il existe une fonction qui permet de faire ça : `incr`, qu'on place devant le nom de la référence. Les trois instructions suivantes sont ainsi équivalentes :

```
quantite := !quantite + 1;
quantite := succ !quantite;
incr quantite;
```

Selon le même principe, on peut `"décrémenter"` une référence avec `decr`. Les trois instructions suivantes sont aussi équivalentes :

```
quantite := !quantite - 1;
quantite := pred !quantite;
decr quantite;
```

`incr` et `decr` permettent donc d'écrire moins de code, et pour cette raison, on les utilisera dès que possible.

2.3.6 Alias et copie

▶ Pouvez-vous essayer de deviner ce qu'il se passe si l'on fait :

```
let s = ref 5 in
let t = s in
...
```

Plus précisément, à quoi le nom `t` sera-t-il associé ?

On se rappelle que dans un `let`, on évalue d'abord l'expression après le `=` et qu'ensuite on associe cette valeur au nom. Appliquons cela. D'abord on crée une boite, contenant 5. Puis on associe le nom `s` à cette boite. Ensuite on évalue l'expression `s`, située entre le `=` et le `in` sur la deuxième ligne : ce `s` est associé à la boite contenant 5, qu'on a créé juste avant. Maintenant, on associe le nom `t` à cette valeur, c'est-à-dire à la boite contenant 5 qu'on a créé au début. Au final, il n'y a qu'une seule boite, et deux noms sont associés à cette boite : `s` et `t`.

★ Afin de vérifier cette conclusion, prolongez le code précédent à l'aide de deux instructions. La première modifiant le contenu de `s`, en lui affectant l'entier 6. La seconde affichant le contenu de `t`. Logiquement, l'ensemble du code doit afficher 6.

_ _

```
let s = ref 5 in
let t = s in
s := 6;
print_int !t;
```

★ Voyons une autre situation, où cette fois-ci on a deux références distinctes. On part de `let box_1 = ref 5 in`. L'objectif est d'écrire les trois instructions suivantes. La première pour créer une référence distincte de la première, et nommée `box_2`. Son contenu initial sera la valeur de `box_1`, non pas en mettant 5, mais en récupérant le contenu de `box_1`. Pour vérifier que les deux boites sont bien distinctes, on affecte alors 7 à `box_1`, avant d'afficher le contenu de `box_2`. L'ensemble du code devrait ainsi afficher 5.

_ _

```
let box_1 = ref 5 in
let box_2 = ref !box_1 in
box_1 := 7;
print_int !box_2;
```

2.4 Exercices

2.4.1 Moyenne et maximum

★ Écrire un programme qui demande à l'utilisateur le nombre de scores qu'il a, puis lui demande ses scores (des entiers positifs) un par un, pour au final afficher la moyenne (un float) de tous ses scores.

_ _

```
let nb_scores = read_int() in
let total = ref 0 in
for score = 1 to nb_scores do
   let valeur = read_int() in
   total := !total + valeur;
done;
let moyenne = (float_of_int !total) /. (float_of_int nb_scores) in
print_float moyenne;
```

Que se passe-t-il s'il n'y a aucun score ? Faîtes le test en tapant 0 comme premier entier, et expliquez l'affichage de `-1.#IND`.

Réponse : il y a une division par zéro au niveau de la déclaration de `moyenne`.

★ Il s'agit de déterminer le meilleur score parmi tous les scores, donnés par l'utilisateur comme précédemment. Ce résultat sera cette fois un entier.

_ _

On va utiliser une boite nommée `meilleur` dont le contenu sera la valeur du meilleur score que l'utilisateur a déjà saisi. La contenu initial sera 0. Il y a alors deux façons de faire. Premièrement, on peut faire un test avec un `if` pour savoir si le score donné est meilleur que celui d'avant :

```
let nb_scores = read_int() in
let meilleur = ref 0 in
for score = 1 to nb_scores do
   let valeur = read_int() in
   if valeur > !meilleur
      then meilleur := valeur;
done;
print_int !meilleur;
```

Remarque : on peut faire le test `(valeur >= !meilleur)`, ça ne change rien au résultat.

Deuxièmement, on part du principe que pour calculer le maximum des `k` premières valeurs, il suffit de prendre le max de la `k-ième` valeur et du maximum des valeurs précédentes. L'idée est de mettre dans le corps de la boucle une instruction qui affecte à la référence nommée `meilleur` le plus grand des deux nombres suivants : `"la nouvelle valeur lue"` et `"du contenu de la référence meilleur"`. Traduction en code :

```
let nb_scores = read_int() in
let meilleur = ref 0 in
for score = 1 to nb_scores do
   let valeur = read_int() in
   meilleur := max !meilleur valeur;
done;
print_int !meilleur;
```

La seconde solution permet de gagner en taille de code et en lisibilité, et est donc préférable.

2.4.2 Affichage de nombres impairs

★ Ecrivez un programme qui affiche tous les nombres impairs entre 21 et 43 (inclus), séparés par des espaces. On fera deux versions :

1. Une boucle sur tous les entiers entre 21 et 43, qui test si le nombre est pair.
2. Une boucle sur tous les entiers entre 10 et 21, qui affiche pour chaque entier son double augmenté de un.

_ _

```
for i = 21 to 43 do
   if i mod 2 = 1 then
      begin
      print_int i;
      print_string " ";
      end;
done;
```

et

```
for i = 10 to 21 do
   print_int (2 * i + 1);
   print_string " ";
done;
```

Dans ce cas la seconde solution se révèle plus efficace.

★ Mais voyons une variante maintenant. Ecrivez un programme qui affiche tous les nombres impairs compris entre deux entiers donnés par l'utilisateur.

_ _

On peut toujours tester facilement quelles valeurs du compteur sont impaires.

```
let initial = read_int() in
let final = read_int() in
for i = initial to final do
   if i mod 2 = 1 then
      begin
      print_int i;
      print_string " ";
      end;
done;
```

Mais il est bien plus compliqué de récupérer les bornes de la boucle dans la seconde méthode. Pour les curieux, voici une méthode qui marche, mais qu'il n'est absolument pas nécessaire de comprendre ni de retenir.

```
let initial = read_int() in
let final = read_int() in
for i = initial / 2 to (final - 1) / 2 do
   print_int (2 * i + 1);
   print_string " ";
done;
```

Keep going ...

Chapitre 3

function

3.1 Principe des fonctions

3.1.1 Objectif

▶ Les boucles `for` permettent de condenser du code qui se répète, à condition que les expressions répétées soient placées les unes à la suite des autres et ne différèrent que par la valeur d'un entier (le compteur). Voici un cas où une boucle `for` ne nous permet pas de condenser le code :

```
for colonne = 1 to 10 do
   print_string "X";
done;

for colonne = 1 to 10 do
   print_string "#";
done;
```

Pourtant, trois lignes sont répétées deux fois dans ce code, à une seule différence près, à savoir ce qui est placé juste après le `print_string`.

Pour mettre en commun tout le code qui est répété, on va utiliser une fonction. Le concept de fonction est extrêmement puissant et fondamental en programmation, et c'est pourquoi on va y consacrer tout le chapitre.

3.1.2 Modèles des robots

▶ Une fonction, *function* en anglais, c'est comme un robot. Lorsqu'on met en marche un robot dans un certain environnement, il effectue les tâches pour lesquelles il a été programmé, et puis il s'éteint lorsqu'il a fini son travail. Une fonction, c'est un robot qui fonctionne de la manière suivante :

- On met le robot en marche en écrivant son nom.
- Le robot commence par ramasser un certain nombre d'objets qu'on a posés devant lui : ce sont ses **arguments** ou **paramètres**.

- Le robot effectue les tâches pour lesquelles il a été programmé. Ces tâches peuvent faire intervenir les arguments et aussi n'importe quel objet de l'environnement dans lequel a été lancé le robot.
- Juste avant la fin, le robot dépose devant lui zéro, un, ou plusieurs objets : ce sont les valeurs qu'il **retourne**.
- Ensuite, le robot s'éteint.

► Prenons des exemples de la vie courante.

1) Un robot qui fait le ménage va :

- ramasser un papier avec le nom de la pièce à nettoyer,
- déposer rien du tout.

2) Un robot qui choisit un disque au hasard dans le placard va :

- ramasser aucun objet,
- déposer un CD.

3) Un robot chargé de trouver tous les fichiers contenant un mot donné va :

- ramasser un mot,
- déposer un certain nombre de noms de fichiers.

► On n'aura pas besoin de robots qui retournent plusieurs objets. En effet, il est toujours possible de retourner une liste d'objets, et cette liste compte alors pour un seul objet. Dans l'exemple 3, si on considère que le robot retourne une liste de noms de fichier, alors il ne retourne qu'un seul objet. Tous nos robots retourneront donc zéro ou un seul objet. Note : on verra plus tard comment gérer des listes d'objets.

3.1.3 Utilisation

► Avant de pouvoir utiliser une fonction, il faut d'abord la **déclarer**, c'est-à-dire donner les plans de fabrication du robot associé. Pour cela, il faut :

- lui donner un nom pour pouvoir l'utiliser plus tard (on nomme la fonction),
- lui expliquer quels objets elle doit ramasser (la fonction **prend des arguments**).
- lui donner les instructions qu'elle doit effectuer (c'est **le corps** de la fonction),
- lui dire les objets qu'elle doit déposer (la fonction **retourne des valeurs**).

Une fois la fonction créée, on pourra **l'appeler** autant de fois que l'on voudra. Pour cela :

- on écrit le nom de la fonction,
- on place des objets séparés par des espaces, qui seront les arguments de la fonction.

Par exemple, si la fonction s'appelle `mafonction` et que l'on veut l'appeler avec pour arguments l'entier `2` et le texte `"ici"`, on écrira pour appeler la fonction :

```
mafonction 2 "ici"
```

▶ Deux cas se présentent alors.

1) Si la fonction ne retourne pas d'objet, l'appel à la fonction est considéré comme une instruction, et on la terminera par un point-virgule :

```
mafonction 2 "ici";
```

2) Si la fonction retourne un objet, on peut nommer cet objet avec un `let` :

```
let x = mafonction 2 "ici" in
```

Ou bien on peut utiliser directement cet objet. Par exemple si l'objet retourné par `mafonction` est un entier, on pourra faire :

```
print_int (mafonction 2 "ici");
```

3.2 Fonctions sans retour

3.2.1 Exemple

▶ Reprenons l'exemple utilisé dans la problématique :

```
for colonne = 1 to 10 do
   print_string "X";
done;

for colonne = 1 to 10 do
   print_string "#";
done;
```

La différence entre les deux codes est juste la chaîne après le `print_string`. Pour faire apparaître clairement que tout le reste du code est identique dans les deux cas, on va réécrire le code en nommant une variable `motif` égale à la chaîne en question.

La première boucle devient :

```
let motif = "X" in
for colonne = 1 to 10 do
   print_string motif;
done;
```

Tandis que la seconde devient :

```
let motif = "#" in
for colonne = 1 to 10 do
   print_string motif;
done;
```

L'idée est d'utiliser une fonction qui va prendre en argument une certaine chaîne nommée `motif`, et qui va effectuer :

```
for colonne = 1 to 10 do
   print_string motif;
done;
```

- `motif` sera l'argument (le paramètre) de la fonction;
- les trois lignes de code précédentes forment le corps de la fonction;
- notez que cette fonction ne retourne rien.

Il ne nous reste plus qu'à trouver un nom pour la fonction.

Pour nommer une fonction, on choisit toujours un nom décrivant précisément l'action réalisée par le corps de cette fonction.

Dans notre cas, `"affiche_dix_fois"` est un bon nom.

▶ Pour définir une fonction, on écrit :

```
let (...nom de la fonction...) (...noms des arguments...) =
   (...instructions du corps de la fonction...)
   in
```

Dans notre exemple, cela donne :

```
let affiche_dix_fois motif =
   for colonne = 1 to 10 do
      print_string motif;
   done;
   in
```

Regardons cela de près :

1. On commence par : `let affiche_dix_fois` pour dire qu'on va associer le nom `affiche_dix_fois` à quelque chose.
2. On écrit le nom du paramètre, ici c'est `motif`.
3. Ensuite, on place le signe égal.
4. On écrit en-dessous les instructions qui forment le corps de la fonction.
5. On termine la déclaration de la fonction avec le mot `in`.

▶ Pour appeler une fonction qui ne retourne rien, on écrit :

```
(...nom de la fonction...) (...valeurs des arguments...) ;
```

On peut donc remplacer :

```
for colonne = 1 to 10 do
   print_string "X";
done;
```

par :

```
affiche_dix_fois "X";
```

En procédant de même pour le reste du code, on remplace au final le source :

```
for colonne = 1 to 10 do
   print_string "X";
done;

for colonne = 1 to 10 do
   print_string "#";
done;
```

par cet autre source :

```
let affiche_dix_fois motif =
   for colonne = 1 to 10 do
      print_string motif;
   done;
   in

affiche_dix_fois "X";
affiche_dix_fois "#";
```

Ces deux sources produisent le même affichage mais ne fonctionnent pas de la même manière.

▶ Quelques petites remarques avant de passer aux exercices.

- Une fonction doit toujours être définie avant de pouvoir être utilisée.
- N'oubliez pas de décaler vers la droite (indenter) le corps de la fonction et le `in`, pour bien visualiser où commence et où se termine la déclaration.
- N'oubliez pas d'ajouter une ligne vide à la suite de la déclaration de la fonction, afin d'aérer convenablement le code.
- On utilisait déjà des fonctions prédéfinies utilisant un paramètre et ne retournant rien. Par exemple `print_int` et `print_float` sont de telles fonctions.

3.2.2 Exercices

★ Utilisez une fonction nommée `ligne_avec_vingt_fois` pour condenser le code suivant :

```
for colonne = 1 to 20 do
   print_string "X";
done;
print_newline();
for colonne = 1 to 20 do
```

```
   print_string "#";
done;
print_newline();
for colonne = 1 to 20 do
   print_string "i";
done;
print_newline();
```

_ _

Il suffit de rajouter l'instruction `print_newline()` dans le corps de la fonction.

```
let ligne_avec_vingt_fois motif =
   for colonne = 1 to 20 do
      print_string motif;
   done;
   print_newline();
   in

ligne_avec_vingt_fois "X";
ligne_avec_vingt_fois "#";
ligne_avec_vingt_fois "i";
```

★ Utilisez une fonction nommée `ligne_x_de_taille` avec le paramètre `nb_colonnes` pour condenser le code suivant :

```
for colonne = 1 to 5 do
   print_string "X";
done;

print_newline();

for colonne = 1 to 7 do
   print_string "X";
done;
```

_ _

Pour bien voir ce qu'il se passe, écrivons l'étape intermédiaire sur la première moitié :

```
let nb_colonnes = 5 in
for colonne = 1 to nb_colonnes do
   print_string "X";
done;
```

On en déduit le code demandé :

```
let ligne_x_de_taille nb_colonnes =
   for colonne = 1 to nb_colonnes do
      print_string "X";
   done;
   in

ligne_x_de_taille 5;
print_newline();
ligne_x_de_taille 7;
```

3.2.3 Fonction avec contexte

▶ On souhaite maintenant utiliser une fonction pour condenser le code suivant :

```
let nb_colonnes = read_int() in

if nb_colonnes > 5 then
   begin
   for colonne = 1 to nb_colonnes do
      print_string "-";
   done;
   end
else
   begin
   for colonne = 1 to nb_colonnes do
      print_string ".";
   done;
   end
;
```

Le code qu'on peut mettre en commun, qui correspond au corps de la fonction, est de la forme :

```
for colonne = 1 to nb_colonnes do
   print_string motif;
done;
```

On va nommer cette fonction `affiche_nb_fois`. Son paramètre est `motif`. Voici la déclaration de la fonction `affiche_nb_fois` :

```
let affiche_nb_fois motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in
```

Pour utiliser cette fonction, on fera `affiche_nb_fois "-"` dans le bloc du `then`, et `affiche_nb_fois "."` dans le bloc du `else`. On aura donc :

```
if nb_colonnes > 5 then
   begin
   affiche_nb_fois "-";
   end
else
   begin
   affiche_nb_fois ".";
   end
;
```

qui se simplifie (attention aux point-virgules) par la réduction des blocs en :

```
if nb_colonnes > 5
   then affiche_nb_fois "-"
   else affiche_nb_fois ".";
```

Pour que le programme fonctionne, il faut précéder ce code de la déclaration de `nb_colonnes` et la déclaration de la fonction :

```
let nb_colonnes = read_int() in

let affiche_nb_fois motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in

if nb_colonnes > 5
   then affiche_nb_fois "-"
   else affiche_nb_fois ".";
```

Et voilà, ça marche !

▶ Expliquez pourquoi le code suivant le compile pas :

```
let affiche_nb_fois motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in

let nb_colonnes = read_int() in

if nb_colonnes > 5
   then affiche_nb_fois "-"
   else affiche_nb_fois ".";
```

Regardons l'erreur de compilation :

```
File "test.ml", line 2, characters 22-33:
Unbound value nb_colonnes
```

La fonction `affiche_nb_fois` utilise la valeur de `nb_colonnes`, et par conséquent le compilateur oblige le programmeur à définir la valeur de `nb_colonnes` avant de déclarer la fonction qui l'utilise. Cela évite d'avoir des situations avec des comportements indéterminés.

3.2.4 Fonction avec lecture

▶ Voyons une variante du code du début. Le programme demande un premier motif à l'utilisateur, et affiche ce motif 10 fois de suite. Ensuite il demande un second motif à l'utilisateur, et l'affiche 15 fois de suite. Voici le code :

```
let motif_1 = read_line() in
for colonne = 1 to 10 do
   print_string motif_1;
done;
print_newline();
let motif_2 = read_line() in
for colonne = 1 to 15 do
   print_string motif_2;
done;
```

Voyez-vous quelle fonction on pourrait utiliser pour condenser ce code ? Cherchez un peu avant de lire la suite. Indication : il faut mettre le `read_line()` à l'intérieur du corps de la fonction.

Le motif répété est :

```
let motif = read_line() in
for colonne = 1 to nb_colonnes do
   print_string motif;
done;
```

La fonction, que l'on appelle `affiche_nb_motifs_choisi`, prend en paramètre `nb_colonnes`. Voici donc le code :

```
let affiche_nb_motifs_choisi nb_colonnes =
   let motif = read_line() in
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in

affiche_nb_motifs_choisi 10;
print_newline();
affiche_nb_motifs_choisi 15;
```

★ Condensez le code suivant à l'aide d'une fonction nommée `passe_hauteur`, avec l'argument `limite` :

```
let hauteur = read_int() in
if hauteur > 10
   then print_string "passe"
   else print_string "bloque";

let hauteur_bis = read_int() in
if hauteur_bis > 18
   then print_string "passe"
   else print_string "bloque";

let hauteur_ter = read_int() in
if hauteur_ter > 7
   then print_string "passe"
   else print_string "bloque";
```

_ _

```
let passe_hauteur limite =
   let hauteur = read_int() in
   if hauteur > limite
      then print_string "passe"
      else print_string "bloque";
   in

passe_hauteur 10;
passe_hauteur 18;
passe_hauteur 7;
```

3.2.5 Plusieurs paramètres

▶ Voyons maintenant une situation où un seul paramètre ne suffit plus. Il s'agit toujours d'afficher une ligne formée de motifs, mais cette fois deux éléments changent : à la fois la longueur de la ligne et aussi le motif qui la forme. Par exemple :

```
for colonne = 1 to 15 do
   print_string "*";
done;

for colonne = 1 to 19 do
   print_string " -_- ";
done;
```

En utilisant `nb_colonnes` et `motif` comme nom des paramètres, on a le corps de la fonction :

```
for colonne = 1 to nb_colonnes do
   print_string motif;
done;
```

On va construire cette fonction à deux paramètres, de la même manière que lorsqu'il y avait un seul paramètre, à ceci près :

- lors de la déclaration de la fonction, on écrira les paramètres à la suite, séparés par des espaces, entre le nom de la fonction et le signe égal,
- lors d'un appel à la fonction, on donnera les valeurs des paramètres dans l'ordre dans lequel ils apparaissent dans la déclaration, et séparés par des espaces.

Dans notre exemple, cela donne :

```
let affiche_ligne_motif nb_colonnes motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in

affiche_ligne_motif 15 "*";
affiche_ligne_motif 19 " -_- ";
```

On résume la gestion des fonctions à plusieurs paramètres en disant que :

Lorsqu'il y a plusieurs paramètres à une fonction, on les place simplement séparés par des espaces.

▶ L'ordre des paramètres qui a été choisi est `nb_colonnes` puis `motif`. Mais ce choix est arbitraire, et l'ordre inverse fonctionne très bien également :

```
let affiche_ligne_motif motif nb_colonnes =
```

Seulement dans ce cas, il est nécessaire d'inverser aussi les paramètres lors des appels :

```
affiche_ligne_motif "*" 15;
affiche_ligne_motif " -_- " 19;
```

★ Écrire une fonction `affiche_rect` qui prend en paramètre un nombre de ligne et un nombre de colonne, et qui affiche un rectangle de ces dimensions rempli de la lettre X.

Pour tester cette fonction, il faut l'appeler avec des paramètres. Première méthode, on écrit directement les paramètres dans le source. Par exemple, des lignes à placer après la définition de la fonction :

```
affiche_rect 8 10;
affiche_rect 15 3;
```

Seconde méthode, on demande à l'utilisateur les paramètres. Pour cela on rajoute le code suivant par exemple :

```
let x = read_int() in
let y = read_int() in
affiche_rect x y;
```

La première méthode s'écrit plus vite, mais la seconde méthode est avantageuse si on veut faire plus de tests.

_ _

```
let affiche_rect nb_lignes nb_colonnes =
   for ligne = 1 to nb_lignes do
      for colonne = 1 to nb_colonnes do
         print_string "X";
      done;
      print_newline();
   done;
   in
```

★ Modifiez la fonction précédente pour qu'elle prenne en troisième paramètre le motif à répéter dans le rectangle. Nommez la nouvelle fonction `affiche_rect_mot`.

Voici par exemple pour tester :

```
affiche_carre 8 10 "X";
affiche_carre 15 3 "I";
```

_ _

```
let affiche_rect_mot nb_lignes nb_colonnes motif =
   for ligne = 1 to nb_lignes do
      for colonne = 1 to nb_colonnes do
         print_string motif;
      done;
      print_newline();
   done;
   in
```

Remarque : le code à l'intérieur de la boucle n'est autre que celui de la fonction `affiche_ligne_motif`. On peut donc écrire si on veut :

```
let affiche_ligne_motif nb_colonnes motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   print_newline();
```

```
    in

let affiche_carre_mot nb_lignes nb_colonnes motif =
    for ligne = 1 to nb_lignes do
        affiche_ligne_motif nb_colonnes motif;
    done;
    in
```

L'intérêt est que si on doit utiliser `affiche_ligne_motif` dans la suite, cette fonction sera déjà définie.

3.2.6 Erreurs

► Lorsque l'ordre des paramètres pour un appel de fonction ne correspond pas à celui donné lors de la déclaration, il peut y avoir un problème :

```
let affiche_ligne_motif motif nb_colonnes =
    for colonne = 1 to nb_colonnes do
        print_string motif;
    done;
    print_newline();
    in

affiche_ligne_motif 15 "*";
```

```
File "test.ml", line 8, characters 20-22:
This expression has type int but is here used with type string
```

L'erreur se situe sur le 15, qui est un entier, et que l'on ne peut pas utiliser comme motif, puisque motif doit être de type string. Le compilateur a en effet déduit du `print_string motif` de ligne 3 que `motif` était de type string.

► Supposons que l'on fasse une erreur lors de la définition de la fonction. Dans l'exemple suivant, on a mis `print_int motif` à la place de `print_string motif` :

```
let affiche_ligne_motif motif nb_colonnes =
    for colonne = 1 to nb_colonnes do
        print_int motif;
    done;
    print_newline();
    in

affiche_ligne_motif "*" 15;
```

Pour le compilateur, il n'y a pas d'erreur dans la déclaration de la fonction : `motif` y est utilisée une seule fois dans un `print_int`, et donc `motif` est supposé être un entier. L'erreur va se situer lors des appels à la fonction :

```
File "test.ml", line 8, characters 20-23:
This expression has type string but is here used with type int
```

L'erreur se situe sur le "*" qui est de type `string`. Comme le compilateur a supposé que `motif` était de type `int`, il s'attendait à trouver un entier et non une chaîne.

3.2.7 Sans paramètre

▶ On a vu des fonctions avec un ou plusieurs paramètres. Dans cette partie, on va s'intéresser aux fonctions qui n'ont aucun paramètre. La fonction effectue alors toujours le même traitement à chaque fois qu'elle est appelée. On a déjà un exemple de telle fonction : `print_newline`, qui affiche à chaque fois qu'on l'appel une nouvelle ligne.

Lorsqu'une fonction n'a aucun paramètre :

- on place un couple de parenthèses `()` entre le nom de la fonction et le signe égal,
- pour réaliser un appel, on place aussi un couple de parenthèses `()` à la suite du nom de la fonction.

On comprendra plus tard pourquoi on met des parenthèses et non pas rien du tout. On résume les deux idées précédentes en disant que :

Lorsqu'il n'y a aucun paramètre à une fonction, on place un couple de parenthèses.

▶ Voici l'exemple d'une fonction nommée `separation` qui affiche un retour à la ligne, puis un trait pointillé, puis un autre retour à la ligne :

```
let separation () =
   print_newline();
   print_string "- - - - -";
   print_newline();
   in
```

Voici un code qui utilise cette fonction :

```
let x = read_int() in
separation();
print_int x;
separation();
print_string "c'est fini !";
separation();
```

★ Ecrivez une fonction `affiche_produit` qui demande à l'utilisateur deux nombres et qui en affiche le produit. Voici un code permettant de tester :

```
print_string "Entrez deux nombres pour la première multiplication :";
print_newline();
affiche_produit();
print_newline();
print_string "Entrez deux nombres pour la seconde multiplication :";
print_newline();
affiche_produit();
print_newline();
```

_ _

```
let affiche_produit () =
   let a = read_int() in
   let b = read_int() in
   print_int (a * b);
   in
```

▶ Il faut faire attention à l'oubli des parenthèses au niveau de la déclaration. Si on les oublie, on obtient une erreur lorsqu'on tente d'utiliser la fonction.

```
let print_separator =
   print_newline();
   print_string "------------------------";
   print_newline();
   in

print_separator();
```

```
File "test.ml", line 7, characters 0-15:
This expression is not a function, it cannot be applied
```

L'erreur est située sur le nom `print_separator`, qui n'est pas une fonction.

Remarque : on ne comprendra que plus tard (lorsqu'on discutera des fonctions plus en détails) pourquoi il n'y a pas eu d'erreur au moment de la déclaration de `print_separator`, alors que l'oubli des parenthèses semble anormal ici.

3.3 Fonctions avec retour

3.3.1 Principe

▶ On va maintenant s'intéresser aux fonctions qui retournent (ou renvoient) une valeur. La robot associé va donc prendre en argument un certain nombre de valeurs, faire des choses avec, et à la fin déposer une valeur de retour.

Le schéma d'une fonction qui retourne une valeur est le suivant :

```
let (...nom de la fonction...) (...noms des paramètres...) =
   (...actions réalisées par la fonction...)
   (...valeur de retour...) in
```

La seule différence par rapport aux fonctions qui ne retournent rien est l'ajout d'une valeur sur la dernière ligne juste avant le `in`.

Certaines fonctions sont simples : elles ne réalisent pas d'actions et se contentent simplement de retourner une valeur calculée à partir des paramètres. On va traiter celles-ci d'abord, on s'occupera des autres ensuite.

3.3.2 Sans actions

▶ Écrivons une fonction qui calcule la moyenne de deux réels. La fonction `moyenne` prend deux arguments réels `a` et `b`, et retourne la moyenne de ces deux nombres qui vaut `(a +. b) /. 2.0`.

```
let moyenne a b =
   (a +. b) /. 2.0 in
```

Maintenant, lorsqu'on écrit quelque part `moyenne x y`, on appelle le robot `moyenne` qui va lire `x` et `y` et déposer la valeur de la moyenne des deux nombres. On peut ensuite faire ce qu'on veut du résultat. Voyons un exemple concret. On peut remplacer le code suivant :

```
let t = (5.3 +. 3.2) /. 2.0 in
print_float t;

print_float( (6.3 +. 7.2) /. 2.0 );
```

par :

```
let t = moyenne 5.3 3.2 in
print_float t;

print_float (moyenne 6.3 7.2);
```

qui est un code qui s'écrit plus facilement.

Voici le programme en entier, avec la définition de la fonction et son utilisation :

```
let moyenne a b =
   (a +. b) /. 2.0 in

let t = moyenne 5.3 3.2 in
print_float t;

print_float (moyenne 6.3 7.2);
```

★ Ecrivez une fonction `carre` qui prend un paramètre réel et qui renvoie son carré.

Pour tester votre fonction affichez `carre 2.5` et vérifiez que vous obtenez bien `6.25`.

_ _

```
let carre x =
   x *. x in
```

★ Ecrivez une fonction `norme_vect_3d` qui prend en paramètre les 3 coordonnées réelles d'un vecteur, et qui renvoie sa norme. Rappel de la formule : $\sqrt{x^2 + y^2 + z^2}$.

_ _

Avec des élévations à la puissance :

```
let norme_vect_3d x y z =
   sqrt (x ** 2. +. y ** 2. +. z ** 2.) in
```

ou bien simplement avec des produits :

```
let norme_vect_3d x y z =
   sqrt (x *. x +. y *. y +. z *. z) in
```

ou même à l'aide de la fonction `carre`, même si cela est peut-être un peu lourd à écrire :

```
let norme_vect_3d x y z =
   sqrt ((carre x) +. (carre y) +. (carre z)) in
```

Pour tester la fonction, il suffit de faire :

```
print_float (norme_vect_3d 2.0 4.0 5.0);
```

et de vérifier que `6.7082039325` est affiché.

3.3.3 Avec actions

▶ On se souvient du code qui permet de calculer la somme des `n` premiers entiers, où `n` est un entier positif quelconque. On fait :

```
let n = ... in
let total = ref 0 in
for i = 1 to n do
   total := !total + i;
done;
```

Et à la fin, `!total` représente la valeur de la somme qu'on veut. L'objectif de cette partie est d'écrire une fonction qui prend en paramètre `n` et qui retourne la somme des `n` premiers entiers. Rappelons le schéma à suivre :

```
let (...nom de la fonction...) (...noms des paramètres...) =
   (...actions réalisées par la fonction...)
   (...valeur de retour...) in
```

Le début de cette fonction sera logiquement :

```
let somme_entiers n =
```

Les actions effectuées par la fonction sont les suivantes :

```
   let total = ref 0 in
   for i = 1 to n do
      total := !total + i;
   done;
```

à savoir la création de la référence, et le calcul à l'aide de la boucle. Quant à la fin de cette fonction, il s'agit de la valeur de retour, c'est-à-dire `!total` suivi d'un `in` :

```
   !total in
```

En mettant les trois morceaux bouts-à-bouts, on obtient la définition de la fonction `somme_entiers` :

```
let somme_entiers n =
   let total = ref 0 in
   for i = 1 to n do
      total := !total + i;
   done;
   !total in
```

Voici un exemple d'utilisation de cette fonction :

```
let resultat = somme_entiers 5 in
print_int resultat;
```

★ Écrire une fonction `factorielle` qui renvoie le produit des `n` premiers entiers, où `n` est un paramètre.

```
let factorielle n =
   let produit = ref 1 in
   for i = 1 to n do
      produit := !produit * i;
   done;
   !produit in
```

Attention, la fonction factorielle dépasse très rapidement la limite des `int`. Si vous rajoutez après la déclaration de la fonction le code :

```
for i = 1 to 13 do
   print_int (factorielle i);
   print_string " ";
done;
```

Vous obtiendrez l'affichage de :

```
1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 -215430144
```

La valeur négative montre qu'on a dépassé la limite des positifs et qu'on est repassé dessous, dans les négatifs.

★ Écrire une fonction nommée `recup_somme` sans paramètre qui, quand on l'appelle, demande deux entiers à l'utilisateur, puis retourne leur somme.

```
let recup_somme () =
   let x = read_int() in
   let y = read_int() in
   x + y in
```

Pour tester, toujours deux solutions :

```
let r = recup_somme() in
print_int r;
```

Ou :

```
print_int (recup_somme());
```

3.3.4 Retour avec `if`

► On peut faire des tests avec des structures en `if` pour déterminer la valeur de retour de la fonction. Premier exemple avec la fonction `maximum`, qui retourne le plus grand des deux paramètres entiers.

```
let maximum a b =
   if a > b
      then a
      else b
   in
```

Pour utiliser cette fonction, c'est toujours le même principe :

```
let x = max 4 6 in
print_int x;
```

Ou même directement :

```
print_int (max 4 6);
```

Remarque : la fonction `maximum` existe déjà par défaut, et s'appelle `max`. Mais on gardera `maximum` pour l'exercice qui suit.

▶ La valeur de retour d'une fonction peut être le résultat d'une structure `if` :

```
let (...nom de la fonction...) (...noms des paramètre...) =
   (...actions de la fonction...)
   if (...condition...)
      then (...valeur de retour 1...)
      else (...valeur de retour 2...)
   in
```

Cette structure fonctionne aussi avec les `else if`.

★ Ecrivez un programme qui demande à l'utilisateur deux entiers, et qui affiche le plus grand des deux. Utilisez pour cela la fonction `maximum` précédemment définie.

- -

```
let maximum a b =
   if a > b
      then a
      else b
   in

let x = read_int() in
let y = read_int() in
print_int (maximum x y);
```

▶ Remarque : la fonction `max` est prédéfinie, on pourra désormais écrire simplement :

```
let x = read_int() in
let y = read_int() in
print_int (max x y);
```

★ Ecrivez une fonction `parite` qui prend en paramètre un entier, et qui retourne la chaîne `"pair"` si le nombre est pair et `"impair"` dans le cas contraire.

- -

```
let parite p =
   if p mod 2 = 0
      then "pair"
      else "impair"
   in
```

Attention, il faut toujours mettre des parenthèses autour de -5 pour éviter que le compilateur essaie de faire : `parite - 5`, c'est-à-dire la soustraction d'une fonction avec un entier !

★ Ecrivez la fonction `absolue` qui retourne la valeur absolue d'un entier donné en paramètre. Cela sans utiliser la fonction prédéfinie `abs`.

```
let absolue s =
   if s >= 0
      then s
      else - s
   in
```

★ Écrire une fonction `plus_grand_carre` qui prend deux entiers en paramètres, qui calcule les carrés de chacun d'entre eux, et retourne la plus grande de ces deux valeurs.

Regardons d'abord l'approche naïve :

```
let plus_grand_carre a b =
   if a * a > b * b
      then a * a
      else b * b
   in
```

C'est bien, ça marche, mais on calcule deux fois les mêmes produits, et par conséquent ce n'est pas joli.

Théoriquement, il serait mieux de poser la valeur des carrés au préalable, mais ça devient un peu long à écrire.

```
let plus_grand_carre a b =
   let ca = a * a in
   let cb = b * b in
   if ca > cb
      then ca
      else cb in
```

En fait, la meilleure solution consiste à utiliser la fonction prédéfinie `max` :

```
let plus_grand_carre a b =
   let ca = a * a in
   let cb = b * b in
   max ca cb in
```

ou encore en faisant `max` directement sur les produits :

```
let plus_grand_carre a b =
   max (a * a) (b * b) in
```

Remarque : il est aussi possible de comparer les valeurs absolues de `a` et `b` avant de renvoyer le carré du plus grand, même si ce n'est pas exactement ce qui était demandé dans l'énoncé.

```
let plus_grand_carre a b =
   let c = max (abs a) (abs b) in
   c * c in
```

3.3.5 Erreurs

▶ Une fonction doit retourner une valeur dans tous les cas. Lorsqu'on utilise une structure `if` pour choisir la valeur de retour, on est obligé d'expliciter le bloc du `else`.

```
let parite n =
   if n mod 2 = 0 then
      "pair"
   else if n mod 2 = 1 then
      "impair"
   in

print_string (parite 3);
```

```
File "test.ml", line 5, characters 11-19:
This expression has type string but is here used with type unit
```

L'erreur est assez déroutante. Elle se situe sur le `"impair"` qui est de type `string`, et qui selon le compilateur devrait être de type `unit`. Le type `unit` est le type `"action"`, comme ce sera expliqué juste après. On peut néanmoins comprendre l'erreur si on se souvient que le `else` est ici implicite. Le code de la fonction est donc équivalent à :

```
let parite n =
   if n mod 2 = 0 then
      "pair"
   else if n mod 2 = 1 then
      "impair"
   else
      ()
   in
```

Le compilateur n'a pas les moyens de voir que le bloc du `else` ne sera jamais exécuté, bien qu'il soit évident pour le programmeur que le nombre est soit pair (`n mod 2 = 0`) soit impair (`n mod 2 = 1`). Le compilateur suppose donc que le bloc du `else` puisse être exécuté. Dans ce cas, le programme fera `()`, c'est-à-dire rien du tout, d'où l'erreur. Et ne renverra pas de valeur (le type `unit` correspond à "ne rien renvoyer", comme on va le voir juste après). Ceci est incompatible avec les valeurs de retour de type `string` que sont `"pair"` et `"impair"`.

3.3.6 Résumé

▶

Le schéma général d'une fonction est le suivant :

```
let (...nom de la fonction...) (...noms des paramètres...) =
   (...actions réalisées par la fonction...)
   (...valeur de retour...) in
```

- si la fonction n'a pas d'argument, on place juste un couple de parenthèses;
- il peut n'y avoir aucune action réalisées par la fonction;
- si la fonction ne retourne rien, il n'y a pas de valeur de retour avant le `in`.
- la valeur de retour peut être le résultat d'un test.

3.4 Typage et notation

3.4.1 Type d'une fonction

▶ Chaque valeur est d'un certain type. Par exemple 5 est un `int`, `"super"` est de type `string`, et `ref 3` est de type `int ref`. Pour les fonctions, c'est pareil, chacune d'elle va avoir un type. Ce type dépendra de deux choses : d'une part des types des paramètres que la fonction prend, et d'autre part du type de valeur que la fonction retourne. Lorsque la fonction ne retourne rien, on utilisera par convention le mot `"unit"` pour traduire cette absence de valeur. Voyons tout de suite une série d'exemples.

▶ La fonction `absolue` :

```
let absolue x =
   if x >= 0
      then x
      else -x
   in
```

prend en paramètre un entier, et renvoie un entier. Le type d'une telle fonction s'écrit par convention de la manière suivante : ce qu'elle prend en paramètre, flèche vers la droite, ce qu'elle donne. C'est pourquoi le type de `absolue` est `" int -> int "`, ce qui se lit "int donne int".

Une petite astuce permet de vérifier que c'est bien le type `int -> int` qui a été déduit par le compilateur pour la fonction `absolue`. L'idée est de faire exprès d'afficher une erreur de typage, de manière à obtenir un message du genre `"this expression has type ... but is here used with type ..."`. Essayons par exemple l'instruction `print_int absolue`, qui est absurde puisqu'une fonction n'est pas un entier, et ne peut donc sûrement pas être affiché avec `print_int`.

```
let absolue x =
   if x >= 0
      then x
      else -x
   in
print_int absolue;
```

```
File "test.ml", line 6, characters 10-17:
This expression has type int -> int but is here used with type int
```

L'expression ligne 6 entre les caractères 10 et 17 est le nom de la fonction `absolue`. Comme prévu, cette expression est de type `int` donne `int`. On a ici essayé d'utiliser `print_int` dessus, qui attend un `int`, et on a donc obtenu une erreur de type.

▶ La règle de typage est exactement la même pour les fonctions qui réalisent des actions avant de renvoyer une valeur :

```
let somme_entiers n =
   let total = ref 0 in
   for i = 1 to n do
      total := !total + i;
   done;
   !total in
```

est aussi de type `" int -> int"`.

▶ Reprenons maintenant la fonction `affiche_ligne` :

```
let affiche_ligne motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   in
```

Cette fonction prend en paramètre une chaîne, nommée `motif`, et ne renvoie rien : elle se contente de faire quelque chose. On a dit que dans ce cas on utilisait le type `"unit"`. Donc le type de la fonction `affiche_ligne` est : `" string -> unit "`.

▶ Le type `"unit"` sert également dans le cas des fonctions qui n'ont pas de paramètres. Ne pas avoir de paramètres est considéré comme avoir un paramètre de type `unit`. Tout simplement :

```
let separation () =
   print_newline();
   print_string "- - - - -";
   print_newline();
   in
```

est de type `" unit -> unit "`.

La fonction `recup_somme` :

```
let recup_somme () =
   let x = read_int() in
   let y = read_int() in
   x + y in
```

est elle de type `" unit -> int "`.

▶ Dernier point, les fonctions à plusieurs paramètres : on met les type des paramètres les uns à la suite des autres, séparés par des flèches, que l'on lit toujours "donne". Et on indique toujours le type de la valeur de retour à la fin. Ainsi :

```
let moyenne a b =
   (a +. b) /. 2.0 in
```

est de type `" float -> float -> float "`.

Autre exemple :

```
let affiche_ligne_motif nb_colonnes motif =
   for colonne = 1 to nb_colonnes do
      print_string motif;
   done;
   print_newline();
   in
```

est de type `" int -> string -> unit "`.

Remarque : il n'y a jamais qu'une seule valeur de retour pour une fonction. Si on veut vraiment retourner plusieurs valeurs d'un coup, il faudra les regrouper dans une structure, et cette structure sera alors considérée comme une seule valeur. Mais on n'étudiera pas cela tout de suite.

3.4.2 Exercices

★ Quels sont les types des fonctions prédéfinies suivantes dont on a écrit les noms :

```
print_newline
print_int
print_float
print_string
read_int
read_float
read_line
```

_ _

Par convention, on écrit le type d'une valeur après le nom de la valeur suivi d'un deux-points.

```
print_newline : unit -> unit
print_int : int -> unit
print_float : float -> unit
print_string : string -> unit
read_int : unit -> int
read_float : unit -> float
read_line : unit -> string
```

★ Quels sont les types de ces autres fonctions prédéfinies :

```
abs           (* valeur absolue d'un entier *)
pred          (* prédécesseur d'un entier *)
abs_float     (* valeur absolue d'un réel *)
sqrt          (* racine carrée d'un réel *)
cos           (* cosinus d'un réel *)
float_of_int  (* réel à partir d'un entier *)
int_of_float  (* entier à partir d'un réel *)
```

_ _

```
abs : int -> int
pred : int -> int
abs_float : float -> float
sqrt : float -> float
cos : float -> float
float_of_int : int -> float
int_of_float : float -> int
```

3.4.3 Polymorphisme

► Certaines fonctions ont des types un peu particuliers. C'est le cas de la fonction `maximum`, qui a la propriété de fonctionner aussi bien avec des paramètres `int` que des paramètres `float` :

```
let maximum x y =
   if x > y
      then x
      else y
   in
```

En regardant l'erreur que nous donnerait une instruction `"print_int maximum;"`, on apprend que la fonction `maximum` a pour type :

```
'a -> 'a -> 'a
```

Le symbole `'a` veut dire : "un type donné". Le type de cette fonction traduit le fait que la fonction doit prendre deux arguments du même type, pour qu'une comparaison puisse être effectuée, et renvoie une valeur du même type que les arguments. Ainsi `maximum` peut être vue comme une une fonction de type `int -> int -> int` ou `float -> float -> float`, mais ne peut pas être vue comme du `int -> float -> float`, par exemple.

On aura l'occasion de reparler de cette propriété d'utilisation de types indéterminés plus tard. On verra que c'est un aspect très puissant de Caml. Pour l'instant, il faut juste savoir que ça existe pour ne pas être dérouté par certains messages d'erreurs.

Well done !

Chapitre 4

array, char, string

4.1 Tableaux

4.1.1 Principe

► La force des ordinateurs est d'être capable de réaliser un très grand nombre de fois des tâches relativement simples. Il peut s'agir de faire un grand nombre de calculs ou encore de traiter une grande quantité de données. Si on doit réaliser une opération sur quelques milliers d'objets de même type, on ne va pas s'amuser à donner un nom différent à chacun d'entre eux. L'idée d'un **tableau**, **array** en anglais, est de donner un nom à ce groupe d'objets, puis de numéroter dans un certain ordre tous les objets qui appartiennent à ce groupe.

On a modélisé une référence comme étant une boite simple dans laquelle on pouvait mettre une unique valeur modifiable. Dans le même esprit, un tableau est une grande boite avec plusieurs compartiments. Chaque compartiment, on dira aussi "case du tableau", est un peu comme une référence : on peut y mettre une valeur, et la modifier si besoin est.

Une propriété importante à vérifier est que tous les compartiments doivent contenir des objets de même type. On ne fabriquera que plus tard des boites avec des compartiments de natures différentes, pouvant accueillir des objets de types différents.

Une référence sur un entier est de type `int ref`, une référence sur un flottant de type `float ref`. Selon la même logique le type d'un tableau d'entier est de type `int array`, un tableau de flottants est de type `float array`, et un tableau de chaîne `string array`. En fait, on peut faire des tableaux contenant des valeurs de n'importe quel type.

Le nombre de compartiments contenus dans le tableau est fixé au moment de sa fabrication. On appelle ce nombre la **taille**, la **longueur**, ou encore la `dimension` du tableau. Il n'est pas possible de changer la taille d'un tableau après sa création. Si on veut augmenter le nombre d'objets numérotés dans le groupe, il faudra fabriquer un autre tableau.

Chaque compartiment est identifié par un numéro unique. L'élément situé dans le compartiment numéro i et dit "élément d'**indice** i". Attention, les numéros commencent à partir de 0 et non à partir de 1. C'est un peu bizarre au début, mais on s'y habitue très vite. Considérons un tableau contenant `taille` compartiments. Le premier élément de ce tableau est donc d'indice 0, le deuxième d'indice 1, le troisième d'indice 2, etc... Enfin, le dernier est d'indice `taille-1` (et non pas d'indice `taille` puisqu'on ne commence pas à 1).

Suivant la même démarche que pour les références, on va commencer par la création des tableaux, puis on passera à leur manipulation. On mettra alors le tout en pratique sur plusieurs exemples.

4.1.2 Création

▶ De même qu'une boite ne peut pas être vide, les compartiments d'un tableau ne peuvent pas être vides. Pour créer un tableau, il faut donc donner les éléments que l'on veut mettre dedans. Pour construire un tableau, on place entre les symboles `[|` et `|]` les éléments du tableau, en les séparant par des points-virgules. Par exemple pour construire un tableau avec des compartiments contenant les entiers 4, 9 et 8 :

```
[| 4; 9; 8 |]
```

- L'élément à l'indice 0 est 4, celui à l'indice 1 est 9, et celui à l'indice 2 est 8.
- Ce tableau est donc de taille 3, et cette taille est définitive pour ce tableau.
- Il contient des entiers, et ne pourra jamais contenir autre chose que des entiers.
- Le type de ce tableau est ainsi `int array`.
- Les valeurs contenues par ce tableau (4, 9 et 8) seront susceptibles d'être modifiées.

Pour nommer ce tableau, il suffit d'utiliser un `let` pour y associer un nom :

```
let mon_tableau = [| 4; 9; 8 |] in ...
```

On a pris comme premier exemple un tableau d'entiers, mais les tableaux fonctionnent aussi bien avec des valeurs de n'importe quel type. Voici un tableau de flottants, donc de type `float array` :

```
let mon_tableau_float = [| 4.32; 3.49; 9.48; 5.33 |] in ...
```

On peut également créer un tableau de chaînes, qui sera ainsi de type `string array` :

```
let mon_tableau_string = [| "un"; "super"; "tableau" |] in ...
```

▶ Cette technique de création ne marche que pour de petits tableaux, puisqu'on ne va pas écrire à la main des milliers de valeurs côtes à côtes. On va donc avoir besoin d'une autre méthode pour construire les tableaux dans des cas plus généraux.

Cette méthode consiste à utiliser une fonction nommée `"Array.make"`, traduit mot à mot `"Tableau.fabrique"`. Ce nom de fonction est particulier, puisqu'il comporte un point au milieu et commence par une majuscule, alors que les points ne sont normalement pas admis dans les noms de variables, et que ceux-ci doivent ne comporter que des minuscules ou des chiffres.

L'explication est la suivante : `Array` correspond à ce qu'on appellera un **module**. C'est une structure qui permet de regrouper un certain nombre de fonctions. L'intérêt des modules est de mettre un peu d'ordre dans les centaines de noms de fonctions utilisables par le programmeur. On retiendra de cela que toutes les fonctions de manipulations de tableaux ont un nom qui commence par `Array` suivit d'un point.

▶ La fonction `Array.make`, qui permet de créer un tableau, s'utilise avec deux paramètres. Le premier est la taille du tableau que l'on souhaite créer. Le seconde paramètre est un objet dont on va mettre une copie dans chacune des cases du tableau que l'on veut créer. C'est une solution simple, qui permet à la fois :

- de préciser le type des compartiments,
- de certifier que tous les compartiments sont de même type,

- de donner un contenu initial à chaque compartiment.

On pourra par la suite modifier le contenu des cases comme on le souhaite.

L'expression `" Array.make taille remplissage "` retourne un tableau de longueur `taille`, dont toutes les cases contiennent la valeur `remplissage`. La longueur d'un tableau est un nombre positif ou nul. La valeur de remplissage peut être de n'importe quel type, et détermine le type du tableau.

Voici par exemple un tableau nommé `t`, de taille 5, et rempli de la valeur 8 :

```
let t = Array.make 5 8 in ...
```

C'est donc un `int array`. Il est équivalent au tableau déclaré élément par élément :

```
let t = [| 8; 8; 8; 8; 8 |] in ...
```

Voici maintenant un tableau nommé `tab_test`, dont toutes les 25 cases contiennent la chaîne `"test"`, et qui est par conséquent de type `string array` :

```
let tab_test = Array.make 25 "test" in
```

Maintenant que le tableau est créé, voyons comment afficher puis changer ses éléments, c'est-à-dire les valeurs contenues dans ses cases.

4.1.3 Lecture

► Si `tab` est un tableau, alors l'expression `tab.(i)` représente la valeur contenue dans sa i-ème case. Plus généralement, pour lire une case dans un tableau, on écrit le nom du tableau, puis un point, et enfin entre parenthèses l'indice de cette case, le tout sans espaces.

Par exemple, après la déclaration suivante de `mon_tableau`

```
let mon_tableau = [| 4; 9; 8 |] in ...
```

l'expression `mon_tableau.(0)` vaut 4, `mon_tableau.(1)` vaut 9, et `mon_tableau.(2)` vaut 8. On peut faire afficher cela par le programme :

```
let mon_tableau = [| 4; 9; 8 |] in
print_int mon_tableau.(0);
print_newline();
print_int mon_tableau.(1);
print_newline();
print_int mon_tableau.(2);
```

On verra un peu plus tard comment récupérer la taille d'un tableau et utiliser une boucle pour afficher tous les éléments d'un tableau, sans recopier ainsi le code.

La lecture fonctionne exactement de la même façon avec les tableaux déclarés à l'aide de `Array.make`. Dans l'exemple suivant, le tableau `tab_test` contient la chaîne `"test"` dans toutes ses cases. On en affiche quelques-unes à l'aide d'un `print_string` pour voir :

```
let tab_test = Array.make 25 "test" in
print_string tab_test.(0);
print_newline();
print_string tab_test.(9);
print_newline();
print_string tab_test.(24);
```

★ Ecrivez un programme qui déclare un tableau nommé `t` constitué de trois cases : la première contient `"ah"`, la deuxième `"eh"`, et la troisième `"oh"`. Affichez alors le contenu de ces trois cases, en commençant par la troisième, puis la première, et enfin la seconde, en séparant ces trois éléments par des espaces.

_ _

```
let t = [| "ah"; "eh"; "oh" |] in
print_string t.(2);
print_string " ";
print_string t.(0);
print_string " ";
print_string t.(1);
```

4.1.4 Modification

► Voici comment modifier le contenu d'une case dont on connaît l'indice dans un tableau : si ce tableau est nommé `tab`, alors l'instruction `tab.(i) <- valeur` permet de mettre la valeur `valeur` dans la case d'indice `i` du tableau. L'ancienne valeur contenue dans cette case est alors définitivement perdue. Plus généralement, pour modifier une case dans un tableau, on écrit le nom du tableau, un point, puis entre parenthèses l'indice de la case, ensuite une flèche vers la gauche, et finalement la nouvelle valeur à mettre dans la case.

Un exemple :

```
let tab = [| 4; 19; -8 |] in
tab.(1) <- 10;
```

Ce code déclare un tableau, et met ensuite la valeur 10 dans la case qui contenait auparavant 19. On peut vérifier que c'est bien ce qu'il se passe en affichant le nouveau contenu de la case d'indice 1 :

```
let tab = [| 4; 19; -8 |] in
tab.(1) <- 10;
print_int tab.(1);
```

Les autres cases (d'indice 0 et d'indice 2) n'ont pas du tout été altérées dans cette manœuvre.

★ Commencez par déclarer le tableau `t` de la manière suivante : `let t = [| "ah"; "eh"; "oh" |] in`, et écrivez ensuite les instructions permettant de mettre `"ih"` dans la première case et `"ah"` dans la dernière. Affichez alors les éléments du tableau, dans l'ordre, et un par ligne.

_ _

```
let t = [| "ah"; "eh"; "oh" |] in
t.(0) <- "ih";
t.(2) <- "ah";
print_string t.(0);
```

```
print_newline();
print_string t.(1);
print_newline();
print_string t.(2);
print_newline();
```

▶ Comme n'importe quel autre type de valeur, les tableaux peuvent être donnés en paramètre d'une fonction. Prenons l'exemple d'une fonction qui affiche le premier élément du tableau d'entiers qui lui est passé en paramètre, puis réalise un retour à la ligne. On teste ensuite cette fonction sur deux tableaux :

```
let affiche_premier tab =
   print_int tab.(0);
   print_newline();
   in

let t1 = [| 4; 6; -3 |] in
affiche_premier t1;

let t2 = Array.make 23 (-2) in
affiche_premier t2;
```

Ce code affiche 4, puis -2. Remarque : cette fonction `affiche_premier` prend en paramètre un tableau d'entiers et ne renvoie rien est par conséquent de type `int array -> unit`.

★ Ecrivez une fonction `somme_deux_premiers` qui prend en paramètre un tableau d'entiers que l'on suppose être de taille deux, et qui retourne la somme des deux éléments de ce tableau. Déterminez au passage le type de cette fonction.

Note: pour fournir un tableau au programme, commencez par écrire sa taille puis ses éléments séparés par des espaces.

_ _

Le type de `somme_deux_premiers` qui prend en paramètre un tableaux d'entiers et qui retourne un entier est `int array -> int`.

```
let somme_deux_premiers tab =
   tab.(0) + tab.(1) in
```

4.1.5 Erreurs de compilation

▶ L'oubli du second paramètre de `Array.make` induit logiquement une erreur. Supposons ainsi qu'on souhaite créer un tableau d'entiers de taille 25, mais qu'on oublie la valeur du contenu initial à mettre dans les cases :

```
let tab = Array.make 25 in
print_int tab.(2);
```

```
File "test.ml", line 2, characters 10-13:
This expression has type 'a -> 'a array but is here used with type 'b array
```

Pour des raisons qu'on comprendra bien plus tard, il n'y a pas d'erreur au moment de la déclaration du tableau. L'erreur ne se situe que lors de son utilisation. Ainsi le nom `tab` ligne 2 pose un problème. Ce nom devrait être

de type `'b array` ce qui veut dire `"tableau d'un type quelconque"`. Pourtant le type de `tab` est `'a -> 'a array` : le fait d'avoir oublié un paramètre à la fonction `Array.make` fait que `tab` a été associé à une fonction ! C'est assez déroutant comme idée pour l'instant, mais cela ne nous empêche pas de repérer le problème. Vu que l'erreur est sur `tab` et que la ligne 2 semble correcte, c'est qu'il faut aller chercher une erreur sur la définition de `tab`. On peut alors s'apercevoir d'un oubli du second paramètre de `Array.make`.

4.1.6 Erreurs d'exécution

► Qu'est-ce que c'est qu'un tableau de taille zéro ? Serait-ce une aberration ? Non, c'est simplement un tableau qui ne contient aucun élément. Quel est l'intérêt d'avoir un tableau vide ? Ca peut très bien arriver dans des cas pratique. Supposons par exemple que l'on utilise un tableau pour stocker les coordonnées des contacts de son carnet d'adresse. Tout au début, il n'y a encore aucun nom dans le carnet. Le tableau associé à ce carnet tout neuf est donc un tableau de taille zéro. Remarque : pour déclarer un tableau vide, on fait par exemple `let t = [||] in...`

► Que se passe-t-il alors si l'on essaie de fabriquer un tableau de taille négative ? Essayons avec `Array.make` de créer un tableau de taille `-3`, rempli de la chaîne `"test"`.

```
let t = Array.make (-3) "test" in
print_string t.(0);
```

On n'obtient pas d'erreur de compilation, l'erreur ne sera détectée qu'au moment de l'exécution :

```
Fatal error: exception Invalid_argument("Array.make")
```

- Une `"exception"` signifie qu'une erreur s'est produite lors d'un appel de fonction.
- `"Fatal error"`, traduire `"Erreur fatale"`, veut dire que le programme a été obligé de s'arrêter brutalement au moment où l'erreur est apparue, car il ne pouvait pas aller plus loin. En effet, il s'est trouvé dans l'incapacité de créer un tableau de taille `-3` (cela n'aurait pas de sens).
- `"Invalid_argument"`, traduire `"Argument invalide"` signifie que c'est un des paramètres fournis lors de l'appel à la fonction qui a été à l'origine du problème.
- `"Array.make"` donne précisément le nom de la fonction qui est à l'origine de l'erreur.

Il est important de remarquer qu'aucune indication sur la ligne du programme où se situe l'erreur ne vous est donnée. Cela est dû au fait que les numéros de lignes ne sont donnés que pour les erreurs de compilation. Pour savoir à quelle ligne est apparue une erreur d'exécution, il faut utiliser un outil spécialisé supplémentaire, appelé `"déboggeur"`.

Le recours à cet outil n'est nécessaire que pour des grands projets de plusieurs milliers de lignes, lorsque le code n'a pas été assez soigné lors de l'écriture. Pour des petits programmes, il est facile de localiser une erreur à l'aide d'un simple coup d'œil au code. Pour les programmes de taille moyenne, de quelques centaines de lignes, nous donnerons à la fin du cours une technique rapide et efficace pour localiser l'erreur.

Pourquoi le compilateur n'est-il pas capable de détecter que le paramètre `-3` va provoquer une erreur ? Simplement parce que dans la plupart des cas il n'est pas possible de prévoir. Que penser du code suivant ?

```
let taille = read_int() in
let tableau = Array.make taille "test" in
print_string "c'est fini";
```

C'est un programme qui terminera correctement que si l'utilisateur fourni un entier positif ou nul. Dans les autres cas une erreur se produira avant la fin.

▶ Voyons maintenant ce qui se passe si on déclare un tableau `let tab = [| 3; 5; -7|] in ...` et qu'on essaie d'accéder à `tab.(-2)` ou `tab.(3)`, c'est-à-dire à des cases du tableau qui n'existent pas.

```
let tab = [| 3; 5; -7|] in
print_int tab.(-2);
```

```
Fatal error: exception Invalid_argument("Array.get")
```

Ce message nous dit qu'il y a eu une erreur fatale à cause d'une exception de type `"Invalid_argument"`, c'est-à-dire d'une erreur générée par un argument invalide dans un appel à une fonction nommée `"Array.get"`, traduire `"Tableau.récupère"`. Quel peut bien être cette fonction `Array.get` qu'on n'a même pas utilisé mais qui provoque pourtant une erreur ?

L'explication est très simple. La notation `tab.(i)` où `tab` est un tableau et `i` est un indice n'est qu'un raccourci pour l'expression : `" Array.get tab i "`. Cet appel de fonction permet exactement de récupérer dans le tableau `tab` l'élément à l'indice `i`. On peut l'utiliser dans son code, c'est juste beaucoup plus lourd que d'écrire simplement `tab.(i)`. Ainsi, les deux lignes suivantes sont totalement équivalentes :

```
print_int (Array.get tab 2);
print_int tab.(2);
```

▶ Le même principe s'applique pour la modification d'une valeur à un indice invalide :

```
let tab = [| 3; 5; -7|] in
tab.(4) <- 12;
```

Sauf que cette fois l'erreur est sur une fonction nommée `"Array.set"` :

```
Fatal error: exception Invalid_argument("Array.set")
```

La fonction `"Array.set"`, traduire `"Tableau.modifier"`, permet de changer le contenu d'une case d'un tableau donné à un indice donné, en précisant la nouvelle valeur lors de l'appel. Ainsi les deux expressions suivantes sont équivalentes :

```
Array.set tab i valeur;
tab.(i) <- valeur;
```

4.1.7 Longueur

▶ `"Array.length"`, traduire `"Tableau.taille"` est une fonction permettant de récupérer la taille du tableau qu'on lui donne en argument. Pour obtenir la taille d'un tableau nommée `tab`, on fait donc simplement `"Array.length tab"`. Un exemple qui affiche la taille, égale à 3, du tableau :

```
let tab = [| 3; 5; -7|] in
print_int (Array.length tab);
```

Remarque : comme la taille d'un tableau est toujours positive ou nulle, le résultat d'un appel à `Array.length` n'est jamais négatif.

Maintenant que l'on sait récupérer la taille d'un tableau, on va pouvoir réaliser des traitements sur l'ensemble des éléments d'un tableau.

► Essayons d'abord d'afficher tous les éléments d'un tableau d'entiers, dans l'ordre et séparés par des espaces. On a déjà vu comment faire :

```
let tab = [| 4; 9; 8; -5 |] in
print_int tab.(0);
print_string " ";
print_int tab.(1);
print_string " ";
print_int tab.(2);
print_string " ";
print_int tab.(3);
print_string " ";
```

Bien évidement, un tel code est inacceptable : il faut utiliser une boucle `for`. Ainsi le code se condense en :

```
let tab = [| 4; 9; 8; -5 |] in
for indice = 0 to 3 do
   print_int tab.(indice);
   print_string " ";
done;
```

Remarquez que le tableau est de taille 4, et que la boucle `for` commence à 0, le premier indice, et se termine à 3, le dernier indice.

★ Ecrivez une fonction `affiche_tableau_entiers` qui prend en paramètre un tableau d'entiers, et qui affiche tous les éléments de ce tableau, dans l'ordre et séparés par des espaces, puis un retour à la ligne à la fin. Testez cette fonction sur divers tableau, y compris un tableau vide (c'est-à-dire de taille zéro). Donnez aussi le type de cette fonction.

– –

C'est une fonction qui prend un tableau d'entiers et fait quelque chose, et qui est par conséquent de type `int array -> unit`.

Les bornes de la boucle for sont : 0 et "taille du tableau moins un". Pour commencer, on pose `taille` le nom du tableau. La valeur finale du compteur est alors `pred taille`, qui on le rappelle est équivalent à `taille - 1`. On fera donc `pred taille` pour définir cette valeur finale. Ce qui donne :

```
let affiche_tableau_entiers tab =
   let taille = Array.length tab in
   for indice = 0 to pred taille do
      print_int tab.(indice);
      print_string " ";
   done;
   print_newline();
   in
```

Pour tester les différents types de tableaux :

```
affiche_tableau_entiers [| 4; 5; -6; 9; 12 |];
affiche_tableau_entiers [||];
affiche_tableau_entiers (Array.make 25 (-2));
```

Remarque : il est bien moins lisible de ne pas déclarer la taille. On évitera d'écrire :

```
let affiche_tableau_entiers tab =
   for indice = 0 to pred (Array.length tab) do
      print_int tab.(indice);
      print_string " ";
   done;
   print_newline();
   in ...
```

En effet, un code compact vers la gauche se lit et se corrige beaucoup plus vite qu'un code avec de longues lignes. On se tiendra donc systématiquement à la règle d'une instruction par ligne.

Lors de l'appel avec le tableau vide, on a `taille = 0` et `pred taille = -1`, et la boucle n'est pas du tout effectuée puisque la valeur finale du compteur est inférieure à sa valeur initiale.

▶ La fonction qu'on vient d'écrire nous servira par la suite. Gardez donc de côté cette fonction que l'on renomme `print_int_array` pour avoir un nom plus pratique :

```
let print_int_array tab =
   for indice = 0 to pred (Array.length tab) do
      print_int tab.(indice);
      print_string " ";
   done;
   print_newline();
   in
```

A partir de maintenant, il faudra aussi conserver dans un fichier à côté toutes les fonctions que vous écrivez, puisqu'on aura souvent besoin de les réutiliser dans la suite.

4.1.8 Résumé

▶ Résumé des manipulations de tableaux :

- Création d'un tableau, première version : `[| elem1; elem2; ... elemN |]`
- Création d'un tableau, seconde version : `Array.make taille motif`
- Accès à l'élément d'indice `i` : `tab.(i)`
- Modification de l'élément d'indice `i` : `tab.(i) <- nouvel_valeur`
- Longueur d'un tableau : `Array.length tab`
- Accès à un indice invalide : `exception Invalid_argument("Array.get")`
- Modification à un indice invalide : `exception Invalid_argument("Array.set")`

4.1.9 Alias et copie

▶ Pour les références, on distinguait la boite (le contenant) de son contenu. Ainsi `b` est une boite, et `!b` est son contenu. Pour les tableaux, ce n'est plus le cas. `tab.(i)` est un compartiment (un contenant) dans lequel on peut mettre des valeurs, mais `tab.(i)` désigne aussi le contenu de ce compartiment.

Alors que se passe-t-il lorsqu'on fait `let x = tab.(i) in` ? Réponse : on associe le nom `x` à la valeur qui est contenue dans `tab.(i)` au moment où l'on réalise cette déclaration. Si après cette déclaration on change le contenu de `tab.(i)`, on ne changera pas la valeur de `x`.

On va l'illustrer par un exemple. On déclare un tableau `tab`, et on associe ensuite le nom `x` au contenu de la case d'indice 0. Ensuite on modifie alors cette case, puis on affiche ensuite la valeur de `x`.

```
let tab = [| 2; 5 |] in
let x = tab.(0) in
tab.(0) <- 3;
print_int x;
```

Ce code affiche alors 2.

En d'autres termes, il n'est pas possible de donner un autre nom à une case du tableau comme on pouvait donner un alias à une référence. Ce qu'on peut faire en revanche, c'est écrire une fonction qui renvoie la valeur de la première case de `tab` au moment où on l'appelle :

```
let tab = [| 2; 5 |] in

let get_tab_0 () =
   tab.(0) in

print_int (get_tab_0 ());
tab.(0) <- 3;
print_int (get_tab_0 ());
```

Ce code affiche donc 2 puis 3.

Remarque : `get_tab_0` est une fonction de type `unit -> int`.

► Comme il était possible de donner plusieurs noms à une même référence, il est possible de donner plusieurs noms à un tableau. Si `tab` est un nom associé à un tableau et que l'on fait `let t = tab in`, alors `t` sera un autre nom associé au tableau.

On vérifie cela en modifiant `t` et en montrant que `tab` est aussi modifié :

```
let tab = [| 2; 5 |] in
let t = tab in
t.(0) <- 3;
print_int tab.(0);
```

affiche 3.

► Pour copier un tableau, c'est-à-dire créer un autre tableau contenant les mêmes valeurs que le premier, il faut utiliser la fonction `Array.copy`.

La fonction Array.copy prend en paramètre un tableau et retourne un autre tableau, copie conforme du premier.

L'exemple suivant associe le nom `t` à une copie du tableau `tab`, et s'assure ensuite qu'une modification sur `t` laisse `tab` inchangé.

```
let tab = [| 2; 5 |] in
let t = Array.copy tab in
t.(0) <- 3;
print_int tab.(0);
```

Ce programme affiche 2.

4.2 Exercices

4.2.1 Tableau des carrés

★ Fabriquez puis affichez un tableau d'entier de taille 30 tel que le contenu de la case d'indice `i` soit le carré de `i`.

_ _

Il est hors de question d'écrire un par un les éléments du tableau : il faut créer un tableau avec `Array.make`, en le remplissant d'une valeur quelconque (par exemple zéro), et utiliser ensuite une boucle `for` pour mettre ce qu'on veut dans chaque case.

```
let tab_carre = Array.make 30 0 in
for i = 0 to 29 do
   tab_carre.(i) <- i * i;
done;
```

`tab_carre` contient alors le tableau que l'on veut. Pour l'afficher, on peut faire :

```
let print_int_array tab =
   for indice = 0 to pred (Array.length tab) do
      print_int tab.(indice);
      print_string " ";
   done;
   print_newline();
   in

let tab_carre = Array.make 30 0 in
for i = 0 to 29 do
   tab_carre.(i) <- i * i;
done;

print_int_array tab_carre;
```

Remarque : il est possible de placer la définition de `print_int_array` après la construction de `tab_carre`.

On ne le mettra plus dans la suite, mais il faut bien sûr déclarer la fonction `print_int_array` avant de pouvoir l'utiliser.

★ Ecrivez maintenant une fonction `fabrique_tab_carre` qui prend en paramètre un entier `taille` et qui retourne un tableau de taille `taille` tel que le contenu de la case d'indice `i` soit le carré de `i`.

_ _

On s'inspire du code d'avant pour écrire le corps de cette fonction. Pour retourner le tableau, il suffit d'en écrire le nom avant le `in` qui termine la déclaration de la fonction.

```
let fabrique_tab_carre taille =
   let tab_carre = Array.make taille 0 in
```

```
   for i = 0 to pred taille do
      tab_carre.(i) <- i * i;
   done;
   tab_carre in
```

On peut tester ce code sur l'exemple suivant :

```
let t = fabrique_tab_carre 30 in
print_int_array t;
```

Mettez cette fonction de côté pour la suite.

4.2.2 Somme des éléments d'un tableau

★ Ecrivez une fonction `somme_tableau_entiers` qui prend en paramètre un tableau d'entiers et qui retourne la somme de tous les entiers qui sont contenus dans ce tableau. Si le tableau est vide, le code doit retourner zéro.

_ _

On utilise une référence pour calculer cette somme.

```
let somme_tableau_entiers tab =
   let somme = ref 0 in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      somme := !somme + tab.(i);
   done;
   !somme in
```

► On va ici voir l'intérêt de garder toutes les fonctions de côté : on peut en combiner l'utilisation.

```
let fabrique_tab_carre taille =
   let tab_carre = Array.make taille in
   for i = 0 to pred taille do
      tab_carre.(i) <- i * i;
   done;
   tab_carre in

let somme_tableau_entiers tab =
   let somme = ref 0 in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      somme := !somme + tab.(i);
   done;
   !somme in

let t = fabrique_tab_carre 30 in
print_int (somme_tableau_entiers t);
```

Ce code affiche la somme des carrées des entiers jusqu'à 30, qui vaut `8555`.

4.2.3 Lecture d'un tableau

★ Ecrivez une fonction nommée `read_int_array` de type `unit -> int array`, qui demande à l'utilisateur une première valeur, qui sera la taille du tableau qu'il souhaite créer, puis qui lui demande la valeur de chacun des éléments du tableau. La fonction doit alors retourner ce tableau.

```
let read_int_array () =
   let taille = read_int() in
   let tab = Array.make taille 0 in
   for i = 0 to pred taille do
      let valeur = read_int() in
      tab.(i) <- valeur;
   done;
   tab in
```

Après avoir copié le code de `print_int_array`, vous pourrez tester votre fonction :

```
let t = read_int_array() in
print_int_array t;
```

Remarque : le corps de la boucle pourrait éventuellement se condenser en `tab.(i) <- read_int()`.

On gardera de côté cette fonction `read_int_array` qui pourra servir plus tard.

4.2.4 Echange de deux valeurs

★ Ecrivez une fonction nommée `echange_cases` qui prend en paramètre trois arguments. Le premier est un tableau `tab` de type quelconque, et les deux autres `i` et `j` sont des indices valides dans ce tableau. Le travail de cette fonction est d'échanger le contenu de la case d'indice `i` avec celui de la case d'indice `j`.

Remarque : cette fonction est de type `'a array -> int -> int -> unit`. En effet, il n'y a pas besoin de connaître le type de contenu des cases du tableau pour échanger deux contenus. La fonction fonctionnera donc pour tous les types de tableau, c'est-à-dire pour tous les tableaux de la forme `'a array`.

Indication : utilisez `print_int_array` pour tester la fonction. Par exemple :

```
let tab = [| 0; 1; 2; 3; 4 |] in
echange_cases tab 2 4;
echange_cases tab 2 0;
print_int_array tab;
```

devra afficher :

```
4 1 0 3 2
```

Ce qu'il ne faut bien sûr absolument pas faire :

```
let echange_cases tab i j =
   tab.(i) <- tab.(j);
   tab.(j) <- tab.(i);
   in
```

car une fois la première opération faite, on a perdu définitivement le contenu de `tab.(i)`, et la seconde instruction se ramènera à mettre dans `tab.(j)` son propre contenu, ce qui n'est pas ce que l'on veut faire.

La solution consiste à nommer `a` et `b` le contenu respectif des deux cases, puis à affecter à ces cases comme nouveau contenu respectivement `b` et `a`.

```
let echange_cases tab i j =
   let a = tab.(i) in
   let b = tab.(j) in
   tab.(i) <- b;
   tab.(j) <- a;
   in
```

On peut économiser une ligne en condensant les deux instructions du milieu. Plutôt que de poser `b` la valeur de `tab.(j)`, puis de mettre la valeur de `b` dans la case `tab.(i)`, on peut directement faire `tab.(i) <- tab.(j)`. Ce qui donne :

```
let echange_cases tab i j =
   let a = tab.(i) in
   tab.(i) <- tab.(j);
   tab.(j) <- a;
   in
```

4.3 Caractères

4.3.1 Écriture

► Une valeur de type `string` est une chaîne de caractères. Comme son nom l'indique, il s'agit d'une juxtaposition de lettres ou de symboles qu'on appelle caractères, et qui forment un texte. Il y a deux catégories de caractères.

► La première catégorie est formée des caractères qu'on peut écrire directement. Ce sont les lettres minuscules et majuscules, les lettres accentuées, les chiffres, l'espace, les signes de ponctuations, ainsi que les opérateurs mathématiques.

```
print_string "abc ABC àéù 123 +-*/%";
```

► La seconde catégorie regroupe des caractères spéciaux. Pour écrire un tel caractère, il faut le précéder d'un `"backslash"`.

a) Le code `\"` permet d'afficher un guillemet, ce qui ne serait pas possible directement puisque ce symbole permet de délimiter les chaînes.

```
print_string "Un texte entre \"guillemets\" comme ceci";
```

b) Le code `\t` représente une tabulation (on l'avait utilisé pour afficher proprement la table de multiplication).

```
print_string "1\t2\t3\t4\t5";
```

c) Le code \n représente un retour à la ligne : `print_string "\n"` réalise la même chose que `print_newline()`.

```
print_string "Multi-ligne :\ndeuxième ligne,\ntroisième ligne.";
```

d) Et comment faire le symbole backslash lui-même ? On en met deux de suite : \\.

```
print_string "Un backslash : \\";
```

★ Voici un texte :

```
print_string "\n" est équivalent à print_newline().
```

Le but est de faire afficher ce texte à votre programme, en utilisant un seul `print_string`.

_ _

Il y a trois caractères spéciaux : les deux guillemets et le backslash. En rajoutant un backslash devant chacun de ces caractères, c'est bon.

```
print_string "print_string \"\\n\" est équivalent à print_newline().";
```

4.3.2 Manipulations

► Un caractère est aussi une valeur à part entière. Son type est `char`, prononcer `"carr"`, pour abréviation de `"character"`. Pour construire une valeur de type `char`, on place entre des guillemets simples ce caractère.

Par exemple : `'a'` est le caractère correspondant à la lettre a. `'\n'` est le caractère de retour à la ligne. `' '` est l'espace. Pour les caractères représentant des caractères spéciaux, il faut les précéder de backslashs. Ainsi : `'\''`, `'\"'`, `'\\'`.

Un caractère est une valeur de type `char` qu'on construit en plaçant le caractère entre des guillemets simples.

► Pour afficher une valeur de type `char`, on dispose de la fonction `print_char`.

★ Affichez avec des `print_char` uniquement le texte suivant :

```
a
b
```

_ _

```
print_char 'a';
print_char '\n';
print_char 'b';
```

► Et pour lire un caractère ? Non, il n'y a pas de fonction `read_char()`. La raison est qu'il est très rare d'avoir besoin de lire qu'un seul caractère, et que de plus il est possible de lire une chaîne avec `read_line()` et d'en récupérer le premier caractère. Ainsi on peut facilement définir soit même une fonction `read_char()` :

```
let read_char () =
   let s = read_line() in
   s.[0] in
```

4.3.3 Erreurs

▶ Les types `string` et `char` étant distincts, on ne peut pas les mélanger. Par exemple :

```
print_char "a";
```

```
File "test.ml", line 1, characters 11-14:
This expression has type string but is here used with type char
```

Erreur sur le `"a"` qui est une chaîne, mais qui devrait être un caractère, vu qu'on a utilisé `print_char`.

On peut aussi avoir le cas inverse :

```
print_string 'a';
```

```
File "test.ml", line 1, characters 13-16:
This expression has type char but is here used with type string
```

▶ L'oubli d'un backslash est fréquent. Essayons d'abord un tel oubli devant un guillemet :

```
print_string "Je m'appelle "arthur" et toi ?";
```

```
File "test.ml", line 1, characters 0-12:
This function is applied to too many arguments,
maybe you forgot a ';'
```

Après `print_string "Je m'appelle "`, le compilateur pense que le texte est terminé, puisque le guillemet délimite le texte. Il s'attend donc a trouver un point-virgule pour finir l'instruction, mais il y a quelque chose après, ce qui le trouble.

On peut aussi oublier un backslash devant un symbole backslash. Supposons que l'on veuille afficher un petit dessin `\-/` par exemple :

```
print_string "\-/";
```

```
File "test.ml", line 1, characters 14-16:
Warning: Illegal backslash escape in string
```

Il y a un problème sur le `\-` que le compilateur considère comme un code spécial, un peu comme il y a `\n` ou `\t`. Mais `\-` ne correspondant à rien du tout, un avertissement est donné. Le code fonctionne tout de même, mais il on ne veut pas laisser de `Warning`, et donc on écrit la version correcte qui est :

```
print_string "\\-/";
```

4.4 Chaînes

4.4.1 Manipulations

▶ Une chaîne de caractères est comme un tableau rempli de caractères. Une chaîne a ainsi une longueur, les caractères qui la composent sont numérotés à partir de zéro, et on peut modifier un caractère donné de la chaîne.

Mais attention, une chaîne, valeur de type `string`, n'est pas traitée comme un tableau de caractères (qui serait de type `char array`). C'est un choix de Caml, puisque d'autres langages assimilent textes et tableaux de caractères. Pour cette raison, il y a des fonctions et des notations spéciales pour les chaînes, qui suivent la même logique que pour les tableaux. Toutes les fonctions de manipulations des chaînes sont regroupés dans le module `String`.

▶ Il y a deux manières de créer une chaîne. La première on la connaît déjà, c'est la construction avec les guillemets :

```
let s = "un texte" in ...
```

La seconde construction se fait avec `String.make`, fonction qui prend en argument deux paramètres. Le premier est la taille de la chaîne à créer, et le second est le caractère qui va servir à remplir la chaîne. Ainsi les deux déclarations suivantes sont équivalentes :

```
let s = String.make 10 'a' in ...
let s = "aaaaaaaaaa" in ...
```

Comme les tableaux vides, il existe les chaînes vides, c'est-à-dire sans aucun caractères. Deux façons de faire :

```
let s = String.make 0 'a' in ...
let s = "" in ...
```

▶ Pour connaître la longueur d'une chaîne, on fait simplement : `String.length` devant le nom de la chaîne.

```
let s = "un texte" in
let t = String.length s in
print_int t;
```

Ce code affiche 8, puisqu'il y a deux lettres pour `"un"`, un pour l'espace, et 5 dans `"texte"`.

▶ Pour accéder le caractère à l'indice `i` dans une chaîne nommée `str`, on fait `str.[i]` avec des crochets et non des parenthèses qui servent pour les tableaux. Par exemple ce code affiche `"uee"` :

```
let s = "un texte" in
print_char s.[0];
print_char s.[4];
print_char s.[7];
```

▶ Pour modifier un caractère, on fait : `str.[i] <- caractère`. Dans le code suivant, on part d'une chaîne `"je teste"`, on modifie le `'t'` qui est à l'indice 3 en un `'r'`, et on affiche le résultat qui est `"je reste"`.

```
let s = "je teste" in
s.[3] <- 'r';
print_string s;
```

▶ Pour copier une chaîne, on utilise `String.copy`, de la même manière que `Array.copy`.

▶ Dernière chose à savoir : on peut mettre bout à bout deux chaînes à l'aide de l'opérateur *accent circonflexe*. Cette opération s'appelle la `"concaténation"`. Ainsi ce code concatène `"début"` et `"fin"` et affiche `"débutfin"` :

```
print_string ("début" ^ "fin");
```

Cet opérateur n'ajoute ni espace ni retour à la ligne entre les deux chaînes. Ainsi pour rajouter un espace entre le code"début" et le `"fin"` pendant notre concaténation, on fait deux concaténation de suite, afin de rajouter un espace au milieu :

```
print_string ("début" ^ " " ^ "fin");
```

ce qui permet d'afficher `"début fin"`.

▶ Résumé des manipulations de chaînes :

- Création d'une chaîne, première version : `"mon texte"`
- Création d'une chaîne, seconde version : `String.make taille un_caractère`
- Longueur d'une chaîne : `String.length str`
- Accès à un caractère : `str.[i]`
- Modification d'un caractère : `str.[i] <- nouveau_caractère`
- Copie d'une chaîne : `String.copy str`.
- Concaténation de deux chaînes : `str1 ^ str2`.

4.4.2 Erreurs

▶ Il ne faut pas confondre les notations des chaînes et des tableaux. Exemple :

```
let str = "test" in
print_char str.(0);
```

```
File "test.ml", line 2, characters 11-14:
This expression has type string but is here used with type 'a array
```

Le `str` est une chaîne, donc de type `string`, et n'est pas un tableau de type `'a array`, comme le laisse entendre la notation avec les parenthèses.

4.4.3 Exercices

★ Ecrivez une fonction `affiche_premiere_et_derniere_lettre` qui prend en paramètre une chaîne et qui en affiche le premier et le dernier caractère, séparés par un espace.

– –

```
let affiche_premiere_et_derniere_lettre str =
   let taille = String.length str in
   print_char str.[0];
   print_char ' ';
   print_char str.[pred taille];
   in
```

Pour tester, par exemple `affiche_premiere_et_derniere_lettre "un test";` affiche `u t`.

Remarque `print_char ' '` et `print_string " "` sont équivalents.

★ La fonction `print_string` permet d'afficher une chaîne. Essayez d'écrire une fonction équivalente, nommée `affiche_chaine` et qui utilise uniquement `print_char` comme fonction d'affichage.

_ _

```
let affiche_chaine str =
   let taille = String.length str in
   for i = 0 to pred taille do
      print_char str.[i];
   done;
   in
```

On peut par exemple tester notre fonction sur : `affiche_chaine "un affichage\n\tréussi";`.

★ Ecrivez une fonction `coupe_premier_et_dernier` qui prend en paramètre une chaîne `str` et qui retourne une chaîne de deux caractères, le premier étant le premier caractère de `str`, et le second étant le dernier caractère de `str`.

_ _

```
let coupe_premier_et_dernier str =
   let taille = String.length str in
   let resultat = String.make 2 ' ' in
   resultat.[0] <- str.[0] ;
   resultat.[1] <- str.[pred taille];
   resultat in
```

Par exemple `print_string (coupe_premier_et_dernier "un test");` affiche `ut`.

Remarque : dans `let resultat = ... in`, on aurait aussi pu utiliser n'importe quelle chaîne de taille 2 à la place du `String.make 2 ' '`. Par exemple `let resultat = "aa" in` fonctionnera tout aussi bien. La différence, c'est que quelqu'un qui lit le code et qui voit 2 comprend que l'objectif est de créer une chaîne de longueur 2. S'il lit `"aa"`, cet objectif lui sera probablement moins explicite. On préférera donc la version avec le `String.make 2 ' '`.

★ Ecrivez un programme qui demande à l'utilisateur trois lignes de textes, et qui déclare la chaîne `total` comme réunion de ses trois textes, qu'on sépare par des espaces. Faites ensuite `print_string total` pour afficher cette valeur.

_ _

```
let s1 = read_line() in
let s2 = read_line() in
let s3 = read_line() in
let total = s1 ^ " " ^ s2 ^ " " ^ s3 in
print_string total;
```

► Une autre solution plus compliquée consiste à utiliser une référence, pour stocker le texte qu'on a déjà lu. Lisez attentivement ce passage, puisqu'il vous aidera pour l'exercice suivant. On crée donc une référence sur une chaîne vide au début. Ensuite, à chaque fois que l'on lit une chaîne, on met cette chaîne à la suite de celle contenu dans la référence (sans oublier l'espace), puis on met le résultat de cette concaténation comme nouveau contenu de la référence. On a donc :

```
let str = ref "" in
let s1 = read_line() in
str := !str ^ s1 ^ " ";
let s2 = read_line() in
str := !str ^ s2 ^ " ";
let s3 = read_line() in
str := !str ^ s3 ^ " ";
print_string !str;
```

Remarque : on a ajouté un espace en plus à la fin, après `s3`, pour que toutes les lignes se ressemblent.

★ Ecrivez un programme qui demande à l'utilisateur combien de chaînes il souhaite donner de lignes (avec un `read_int()`), qui demande à l'utilisateurs ces chaînes une par une (avec un `read_line()`). Le but est de construire une chaîne qui est la concaténation de toutes les chaînes données par l'utilisateur, ces chaînes étant séparées par des espaces. Afficher à la fin cette chaîne (avec un seule `print_string`).

_ _

L'idée est d'utiliser une référence qui contient la concaténation des textes qu'on a déjà lus. A chaque lecture d'un nouveau texte, on le concatène au contenu de la référence. Pour répéter ces lectures, on utilise une boucle `for`.

```
let nb = read_int() in
let str = ref "" in
for i = 1 to nb do
   let ajout = read_line() in
   str := !str ^ ajout ^ " ";
done;
print_string !str;
```

Remarque : on pourrait faire un test avec un `if` pour éviter l'espace en trop à la fin.

4.4.4 Nombre en chaîne

► On peut convertir un nombre entier en la chaîne qui le représente à l'aide de la fonction `string_of_int`.

Ainsi `53` est un entier, et `string_of_int 53` est la chaîne `"53"`. Cette chaîne peut être manipulée comme toutes les autres.

Exemple d'un affichage de l'entier 53 :

```
let str = string_of_int 53 in
print_string str;
```

En fait, `print_int 53` est équivalent à `print_string (string_of_int 52)`.

★ Ecrivez une fonction nommée `detail_signe` qui prend en paramètre un entier `p` et qui retourne une chaîne :

- si `p` est positif, la fonction doit retourner `"Le nombre p est positif"`, en remplaçant `p` par sa valeur;
- si `p` est négatif, la fonction doit retourner `"Le nombre p est négatif"`, en remplaçant `p` par sa valeur;
- si `p` est nul, la fonction doit retourner `"Le nombre 0 est nul"`.

Par exemple, si `p = 53`, la fonction renvoie : `"Le nombre 53 est positif"`.

_ _

Première solution : on teste selon les différents cas :

```
let detail_signe p =
   if p > 0 then
      "Le nombre " ^ (string_of_int p) ^ " est positif"
   else if p < 0 then
      "Le nombre " ^ (string_of_int p) ^ " est négatif"
   else
      "Le nombre 0 est nul"
   in
```

Version améliorée : on met en commun le début de la chaîne à renvoyer, ce qui évite de recopier du code.

```
let detail_signe p =
   let debut = "Le nombre " ^ (string_of_int p) ^ " est " in
   if p > 0
      then debut ^ "positif"
   else if p < 0
      then debut ^ "negatif"
   else
       debut ^ "nul"
   in
```

Remarque : on verra bien plus tard une version encore mieux, mais on n'a pas les moyens de l'expliquer maintenant.

▶ Il existe l'équivalent de `string_of_int` pour les flottants : c'est `string_of_float`.

`string_of_float` permet de convertir un nombre flottant en la chaîne qui le représente.

Ainsi, `print_float x` est équivalent à `print_string (string_of_float x)`.

4.5 Exercices

4.5.1 Symétrisation d'une chaîne

★ Écrire une fonction `inverse_chaine` qui prend une chaîne en paramètre et en renvoie son miroir, c'est-à-dire que la première lettre se retrouve la dernière, la seconde lettre se retrouve l'avant-dernière, etc... tandis que la dernière vient prendre la première place. Par exemple, pour la chaîne `"Vive Caml"`, la fonction doit renvoyer une chaîne contenant `"lmaC eviV"`.

Pour tester votre code, vous pouvez d'abord essayer :

```
print_string (inverse_chaine "Vive Caml");
```

puis pour faire plus de tests :

```
let chaine = read_line() in
print_string (inverse_chaine chaine);
```

_ _

Appelons `entree` la chaîne fournie en paramètre, et `sortie` la chaîne que l'on retournera. Ces ceux chaînes sont de même longueur, on nommera cette valeur `taille`. Nous allons voir deux solutions au problème, qui sont assez différentes l'une de l'autre, et toutes les deux intéressantes.

▶ La première repose sur l'analyse suivante.

- On veut affecter au premier caractère de la chaîne `sortie`, désigné par `sortie.[0]`,la valeur du dernier caractère de `entree`, soit `entree.[taille-1]`.
- De même, pour le deuxième caractère, on veut donner à `sortie.[1]` la valeur de `entree.[taille-2]`.
- Puis à `sortie.[2]` la valeur de `entree.[taille-3]`.
- Au bout de la chaîne, le dernier caractère de `sortie`, nommé `sortie.[taille-1]` doit être égal au premier caractère de `entree`.
- En généralisant, si `i` est un indice de position dans la chaîne `sortie`, on veut donner à `sortie.[i]` la valeur de `entree.[taille-1-i]`.
- Pour se convaincre que cette formule est la bonne, on vérifie sa validité aux bornes, c'est-à-dire lorsque `i = 0` et lorsque `i = taille-1`. Si la formule marche pour les deux extrémités, il y a de fortes chances qu'elle marche tout le temps.

On commence par construire une chaîne nommée `sortie` de la même taille que la chaîne d'entrée. On fixe alors un par un les caractères de la chaîne `sortie` à l'aide d'une boucle `for`, dont le corps est formé de l'instruction `sortie.[i] <- entree.[taille - 1 - i]`. A la fin, on retourne la chaîne `sortie`.

```
let inverse_chaine entree =
   let taille = String.length entree in
   let sortie = String.make taille 'a' in
   for i = 0 to taille - 1 do
      sortie.[i] <- entree.[taille - 1 - i];
   done;
   sortie in
```

▶ La seconde solution consiste à intervertir deux par deux les lettres qui doivent échanger de place. On commence déclarer `sortie` comme étant une copie de `entree`, et on modifier ensuite `sortie`. D'abord on échange le premier caractère avec le dernier. Ensuite on échange le second avec l'avant-dernier. Puis le troisième avec l'avant-avant-dernier. Où s'arrête-t-on ? Au milieu.

Il y a deux possibilités pour le milieu. Si la chaîne est de longueur impaire, il va y avoir un caractère tout seul au milieu qui restera en place. Si la chaîne est de longueur paire, les deux caractères les plus au milieu devront être échangés.

Combien d'échanges faut-il donc réaliser ? Si `taille` est pair, il faut en faire `taille / 2`. Si taille est impair, il faut en faire `(taille - 1) / 2`. On peut généraliser cela à l'aide de la division entière en `taille / 2`. Le compteur `i` de la boucle `for` dont le corps échange `sortie.[i]` et `sortie.[taille - 1 - i]` commence à `i = 0` et se termine à `i = pred(taille / 2)`, de manière à avoir réalisé `taille / 2` échanges.

```
let inverse_chaine entree =
   let sortie = String.copy entree in
   let taille = String.length entree in
   for i = 0 to pred (taille / 2) do
      let temp = sortie.[i] in
      sortie.[i] <- sortie.[taille - 1 - i];
      sortie.[taille - 1 - i] <- temp;
   done;
   sortie in
```

▶ Que se passerait-il si on modifiait directement la chaîne `entree`, c'est-à-dire si on n'en faisait pas une copie pour faire les modifications ?

```
let inverse_chaine_bis chaine =
   for i = 0 to pred (taille / 2) do
      let temp = chaine.[i] in
      chaine.[i] <- chaine.[taille - 1 - i];
      chaine.[taille - 1 - i] <- temp;
   done;
   chaine in

let s = "un test" in
let t = inverse_chaine_bis s in
print_string t;
```

La valeur retourne un résultat correct, mais la chaîne qui est retourné est la même que celle qui a été donné en paramètre. C'est-à-dire que `s` et `t` sont les mêmes chaînes. Pour s'en rendre compte, rajoutez le code suivant, qui affiche `s`, et qui regarde si une modification sur `t` modifie également `s`.

```
print_string s;
t.[0] <- 'r';
print_string s;
```

affiche `"tset nu"` puis `"rset nu"`.

En fait, la fonction `inverse_chaine_bis` n'a pas besoin de retourner une chaîne. Écrivons la fonction `inverse_chaine_ter`, de type `string -> unit`, qui prend une chaîne en paramètre et qui remplace cette chaîne par son symétrique :

```
let inverse_chaine_ter chaine =
   for i = 0 to pred (taille / 2) do
      let temp = chaine.[i] in
      chaine.[i] <- chaine.[taille - 1 - i];
```

```
      chaine.[taille - 1 - i] <- temp;
   done;
   in

let s = "un test" in
inverse_chaine_ter s;
print_string s;
```

Vocabulaire : on dit que la fonction `inverse_chaine_ter` modifie `chaine` **en place**.

Il est important de bien saisir la différence :

- la fonction `inverse_chaine` ne modifie pas son argument, et renvoie une autre chaîne contenant le résultat,
- la fonction `inverse_chaine_ter` modifie son argument et ne renvoie rien.

Ces deux fonctions sont donc conceptuellement différentes, même si on utilise l'une où l'autre pour faire la même tâche, qui est de retourner une chaîne. Y a-t-il une fonction mieux que l'autre ? Non, cela dépend entièrement de ce que l'on veut en faire. On reviendra sur cette question.

4.5.2 Concaténation de tableaux

★ On a vu qu'il y avait un opérateur permettant de mettre bout-à-bout deux chaînes de caractères. Il y a l'équivalent pour les tableaux : la fonction `"Array.append t1 t2"` permet de concaténer deux tableaux de même type (pour que cela ait un sens). Elle retourne un nouveau tableau, dont la longueur est la somme des longueurs de `t1` et de `t2`.

```
[|         t 1          |] [|              t 2              |]
-------------------------------------------------------------
[|              r   e   s   u   l   t   a   t                 |]
```

Par exemple si `t1 = [| 3; 5 |]` et `t2 = [| 4; 9; 0 |]`, alors `Array.append t1 t2` retournera le tableau `[| 3; 5; 4; 9; 0 |]`.

C'est un très bon exercice que d'essayer de recoder cette fonction soit même. Ecrivez donc une fonction `concatene_tab` qui fait la même chose que `Array.append`. Pouvez-vous en deviner le type de cette fonction ?

Indication pour le code : pour créer le tableau que l'on retourne, il faut en donner une valeur d'initialisation. Pour cette exercice, on supposera que le tableau `t1` n'est pas vide, et on pourra donc prendre `t1.(0)` pour initialiser le tableau `resultat`. Sans cette supposition, le code serait un peu plus compliqué car il faudrait faire des tests.

– –

► La fonction `concatene_tab` prend en paramètre deux tableaux de même type et renvoie un tableau du même type. Son type est donc `'a array -> 'a array -> 'a array`.

Soit `s1` et `s2` les longueurs respectives des deux tableaux. Le tableau final, appelé `resultat`, est de longueur `s1 + s2`, et son dernier élément se situe donc à l'indice `s1 + s2 - 1`. Récapitulons les indices des extrémités des tableaux sur le schéma correspondant aux trois tableaux :

```
[|0   1   ...   (s1-1)|] [|0   1       ...       (s2-1)   |]
-------------------------------------------------------------
[|0   1   ...   (s1-1)    s1  (s1+1)    ...     (s1+s2-1)|]
```

Le début du tableau `resultat` est identique au premier tableau (`t1`), donc pour tout indice `i` strictement inférieur à `s1`, on veut `t1.(i) = resultat.(i)`. D'autre part la fin du tableau `resultat` est identique au second tableau (`t2`). Ainsi, le premier élément de `t2` va se retrouver à l'indice `s1` dans `resultat`. Pour le deuxième tableau, il faut donc décaler les indices d'une quantité `s1` : pour tout indice `i` strictement inférieur à `s2`, `t2.(i) = resultat.(i + s1)`.

Il est bon de vérifier ce qu'il se passe au bout du tableau. Pour la dernière valeur du compteur dans la seconde boucle (`i = s2 - 1`), l'instruction effectuée : `resultat.(s1 + s2 - 1) <- t2.(s2 - 1)`. Cela vérifie que le dernier élément de `t2` arrive dans le dernier compartiment de `resultat`.

```
let concatene_tab t1 t2 =
   let s1 = Array.length t1 in
   let s2 = Array.length t2 in
   let resultat = Array.make (s1 + s2) t1.(0) in
   for i = 0 to pred s1 do
      resultat.(i) <- t1.(i) ;
   done;
   for i = 0 to pred s2 do
      resultat.(i + s1) <- t2.(i);
   done;
   resultat in
```

► Dans le code précédent, on utilise des compteurs qui travaillent sur les indices valides dans `t1` et `t2`. En effet on fait `t1.(i)` et `t2.(i)`. Une variante consiste à utiliser des indices valides dans `resultat`, en faisant toujours `resultat.(i)`. La première boucle `for` n'est pas modifiée, mais la seconde l'est : on rajoute une quantité `s1` à la valeur du compteur.

```
   for i = 0 to pred s1 do
      resultat.(i) <- t1.(i);
   done;
   for i = s1 to pred (s1 + s2) do
      resultat.(i) <- t2.(i - s1);
   done;
```

Maintenant, on a une boucle `for` pour des indices entre 0 et `pred s1`, et une autre entre `s1` et `pred (s1 + s2)`. Les corps se ressemblent mais ne sont pas tout à fait pareils. Si on veut utiliser une seule boucle, c'est possible, il suffit de mettre un test sur la valeur du compteur dans le corps de la boucle :

```
let concatene_tab t1 t2 =
   let s1 = Array.length t1 in
   let s2 = Array.length t2 in
   let resultat = Array.make (s1 + s2) t1.(0) in
   for i = 0 to pred (s1+s2) do
      if i < s1
         then resultat.(i) <- t1.(i)
         else resultat.(i) <- t2.(i - s1);
   done;
   resultat in
```

► Les deux méthodes présentées correspondent à deux façons de différentes de voir le problème. On ne peut pas vraiment dire qu'il y en ait une mieux que l'autre. De toutes manières, en pratique, on utilise directement `Array.append`.

That's cool !

Chapitre 5

bool, random

5.1 Valeurs booléennes

5.1.1 Constructions

▶ Dans les structures conditionnelles qu'on a eu jusqu'à maintenant, il y avait toujours un simple test. Exemple :

```
let x = read_int() in
if x > 5
   then print_string "supérieur"
   else print_string "inférieur";
```

La condition est ici `x > 5` entre le `if` et le `then`. Le résultat de ce test est soit vrai, soit faux. En anglais, on dit `true` ou `false`. Une valeur de cette nature est dite `"booléenne"`, et il y correspond le type `"bool"`.

▶ Une valeur booléenne est une valeur comme une autre. Rien n'empêche de lui donner un nom. Dans cet exemple, on appelle `b` le booléen associé à la réponse à la question `"est-ce que x est strictement plus grand que 5 ?"`, qui s'écrit en Caml `(x > 5)`.

```
let x = read_int() in
let b = (x > 5) in
...
```

Remarque : on a placé le test `x > 5` entre parenthèses. C'est un choix de présentation. Pourquoi des parenthèses ? Supposez que l'on souhaite tester si x vaut 5 ou non. Sans les parenthèses, cela donnerait `let b = x = 5 in ...` ce qui serait visuellement très déroutant, à cause de la présence de plusieurs signes d'égalité qui ont des significations différentes.

▶ On peut aussi utiliser directement les valeurs `true` et `false` pour construire un booléen. Exemple :

```
let b = true in ...
```

Une valeur booléenne (`bool`) vaut `true` ou `false`, et peut se construire avec des tests comparatifs.

▶ Revenons aux structures de décision :

La condition d'un `if` est toujours une expression de type `bool`.

Voici un programme équivalent au premier du chapitre. La seule différence est qu'on a posé `b` la valeur booléenne `(x > 5)`. Ce booléen `b` détermine ensuite le comportement du `if`.

```
let x = read_int() in
let b = (x > 5) in
if b
   then print_int "supérieur"
   else print_int "inférieur";
```

Que pensez-vous du code suivant ?

```
if false
   then print_int "ahh"
   else print_int "ohh";
```

Réponse : il est équivalent à `print_int "ohh";`, puisque c'est toujours le bloc du `else` qui est effectué.

★ Ecrivez une fonction nommée `est_pair` qui prend un entier `n` en paramètre et renvoie `true` lorsque celui-ci est pair, et `false` sinon. Trouvez ensuite un moyen de tester votre fonction.

– –

`n` est pair lorsque `n mod 2` vaut 0. Le booléen correspondant à la question `"est-ce que x est pair ?"` est `((n mod 2) = 0)`. C'est cette valeur que va retourner la fonction `est_pair` :

```
let est_pair n =
   (n mod 2 = 0) in
```

Pour tester la fonction, on peut soit nommer son résultat et l'utiliser comme condition d'un test :

```
let result = est_pair 3 in
if result
   then print_string "pair"
   else print_string "impair";
```

soit mettre directement l'appel à la fonction après le `if` :

```
let est_pair n =
   (n mod 2 = 0) in
if est_pair 3
   then print_string "oui"
   else print_string "non";
if est_pair 4
   then print_string "oui"
   else print_string "non";
```

Qui affiche logiquement `"non"` puis `"oui"`.

5.1.2 Affichage

► Pour afficher la valeur de retour d'une fonction qui renvoie un booléen, on aimerait bien disposer d'une fonction permettant d'afficher un booléen, c'est-à-dire une fonction prenant en paramètre un booléen et qui affiche `"true"`

ou `"false"` selon la valeur de ce booléen. La fonction `print_bool` n'est pas une fonction prédéfinie, car elle n'est pas d'un usage très fréquent. Heureusement, on peut la définir soi-même facilement.

★ Ecrivez une fonction `print_bool` à l'aide d'un test `if`. Quel est le type de cette fonction ?

_ _

La fonction est de type `bool -> unit`.

```
let print_bool b =
   if b
      then print_string "true"
      else print_string "false";
   in
```

Pour tester, on peut par exemple faire `print_bool true`, qui affiche `"true"`, puis `print_bool (3 > 4)`, qui affiche `"false"`.

► La fonction `string_of_bool`, traduire `"chaîne à partir d'un booléen"`, est une fonction prédéfinie. Elle prend en paramètre un booléen et retourne sa représentation sous forme d'une chaîne `"true"` ou `"false"`. Cette fonction a été définie par :

```
let string_of_bool b =
   if b
      then "true"
      else "false"
   in
```

On peut utiliser cette fonction prédéfinie pour afficher facilement un booléen. Par exemple, si `b` est un booléen qui vaut vrai, alors l'instruction suivante affiche `"true"`.

```
print_string (string_of_bool b);
```

Si on a besoin d'afficher plusieurs booléens, on pourra définir `print_bool` de la manière suivante :

```
let print_bool b =
   print_string (string_of_bool b);
   in
```

► Pour afficher des booléens, on définit généralement la fonction `print_bool` au préalable :

```
let print_bool b =
   print_string (string_of_bool b);
   in
```

Par exemple, pour tester le code de la fonction `est_pair` :

```
let print_bool b =
   print_string (string_of_bool b);
   in

let est_pair n =
   (n mod 2 = 0) in
```

```
print_bool (est_pair 3);
print_string " ";
print_bool (est_pair 4);
```

Ce code affiche `"false true"`.

5.1.3 Lecture

▶ Pour lire un entier, il n'y a pas de fonction `read_bool()`. Là encore parce qu'elle ne sert pas beaucoup. Mais il y a la fonction `bool_of_string`, traduire `"booléen à partir d'une chaîne"`. Ainsi on peut définir soi-même `read_bool` :

```
let read_bool() =
   let s = read_line() in
   bool_of_string s in
```

L'utilisateur doit taper `"true"` ou `"false"` fournir le booléen.

★ En utilisant les fonctions `string_of_bool`, écrivez un programme qui demande deux booléens à l'utilisateur, et qui affiche s'ils sont égaux ou non.

– –

```
let read_bool() =
   let s = read_line() in
   bool_of_string s in

let b1 = read_bool() in
let b2 = read_bool() in
print_string (string_of_bool (b1 = b2));
```

▶ En pratique, il est beaucoup plus simple de lire un entier. L'association généralement utilisée est zéro pour `false` et n'importe quel autre entier pour `true`. Voici la traduction immédiate en code :

```
let read_bool() =
   let x = read_int() in
   if x = 0
      then false
      else true in
```

Cette fonction peut être condensée en :

```
let read_bool() =
   (read_int() <> 0) in
```

En effet, on renvoie bien vrai dès que l'entier lu est différent de 0.

On peut réutiliser le code de l'exercice précédent pour tester notre nouvelle fonction. Il suffit maintenant à l'utilisateur de taper 0 pour faux, et 1 (ou tout autre entier) pour vrai.

```
let b1 = read_bool() in
let b2 = read_bool() in
print_string (string_of_bool (b1 = b2));
```

5.1.4 Exercices

★ Simplifiez la fonction suivante :

```
let passe_ou_non x =
   if x < 10
      then false
      else true
   in
```

_ _

La fonction renvoie une valeur booléenne qui est vraie si `x` est supérieur ou égal à 10. Donc :

```
let passe_ou_non x =
   (x >= 10) in
```

★ Ecrivez une fonction qui analyse la performance d'un concurrent dans une course de la manière suivante. La fonction `est_performant` prend en paramètres le nombre total de participants et le rang d'arrivée. Elle renvoie vrai si le concurrent se situe strictement dans la première moitié des participants (c'est-à-dire renvoie faux si le candidat arrive juste au milieu ou au-delà). Quel est le type de cette fonction ?

_ _

Type : `int -> int -> bool`.

On peut faire un test pour discuter selon que le nombre de participant est pair ou impair.

```
let est_performant participants rang =
   if (participants mod 2) = 0
      then (rang <= participants / 2)
      else (rang <= (participants - 1) / 2)
   in
```

Mais on peut aussi traiter d'un coup les deux parités grâce à la division entière.

```
let est_performant participants rang =
   (rang <= participants / 2) in
```

Voyez-vous comment modifier la fonction si on veut cette fois renvoyer vrai aussi lorsque le candidat arrive juste au milieu ?

Réponse :

```
let est_performant_bis participants rang =
   (rang <= (participants + 1) / 2 ) in
```

5.2 Calcul booléen

5.2.1 ET logique

► On peut avoir besoin de construire des conditions plus élaborées dans les tests. Voici un exemple : on souhaite écrire un programme qui demande à l'utilisateur un âge (entier) et qui affiche `"adolescent"` si cet âge est compris

entre 12 (inclus) et 18 ans (exclus), et `"autre"` sinon.

La condition d'âge supérieur à 12 ans se traduit par `(age >= 12)` et celle d'âge inférieur à 18 ans par `(age < 18)`. Pour dire qu'on veut avoir les deux conditions en même temps, on place entre elles le symbole `&&` qui se lit `"et"`. D'où le code :

```
let age = read_int() in
if (age >= 12) && (age < 18)
   then print_string "adolescent"
   else print_string "autre";
```

Le code `"(age >= 12) && (age < 18)"` forme une **expression booléenne**. On peut utiliser cette expression comme valeur de retour d'une fonction aussi. Par exemple :

```
let est_adolescant age =
   (age >= 12) && (age < 18) in

let age = read_int() in
let b = est_adolescant age in
print_string (string_of_bool b);
```

★ Écrire une fonction `taille_convenable` qui prend en paramètre la largeur puis la longueur d'une pièce (sous forme de nombres entiers de centimètres) et qui renvoie vrai si trois conditions sont réunies. D'une part la longueur doit être supérieure à 400 et la largeur supérieure à 300, et d'autre part la différence entre ces deux valeurs ne doit pas dépasser 200. Toutes ces inégalités peuvent être des égalités. Remarque : on suppose que la longueur est toujours supérieure à la largeur donnée.

_ _

```
let taille_convenable largeur longueur =
   (longueur >= 400) && (largeur >= 300) && ((longueur - largeur) <= 200) in
```

Pour tester, vérifiez que ce code affiche `"true"` :

```
print_string (string_of_bool (taille_convenable 400 600));;
```

★ Écrire une fonction `est_multiple_trois_et_cinq` qui teste si un nombre est multiple à la fois de 3 et de 5 :

_ _

```
let est_multiple_trois_et_cinq n =
   (n mod 3 = 0) && (n mod 5 = 0) in
```

Mais être multiple de 3 et de 5 à la fois est équivalent à être multiple de 15, donc la même fonction peut s'écrire de manière plus condensée :

```
let est_multiple_trois_et_cinq n =
   (n mod 15 = 0) in
```

5.2.2 OU logique

▶ On vient de voir les expressions booléennes contenant un ET, maintenant on va voir celles qui contiennent un OU. Le symbole pour le OU est `||`.

Ce code affiche `"tarif réduit"` si l'âge donné est plus petit que 18 ou bien plus grand que 65, et `"plein tarif"` dans les autres cas.

```
let age = read_int() in
if (age < 18) || (age >= 65)
   then print_string "tarif réduit"
   else print_string "plein tarif";
```

Remarque : on aurait pu tester la condition contraire. Dans ce cas, on aurait utilisé un `ET` :

```
let age = read_int() in
if (age >= 18) && (age< 65)
   then print_string "plein tarif"
   else print_string "tarif réduit";
```

★ Écrire un programme qui lit un prénom, et qui affiche `"c'est moi ou MOI"` si ce prénom est le vôtre tout en minuscules ou bien tout en majuscules. Affichez `"quelqu'un d'autre"` dans les autres cas.

– –

```
let prenom = read_line() in
if (prenom = "arthur") || (prenom = "ARTHUR")
   then print_string "c'est moi ou MOI";
   else print_string "quelqu'un d'autre"
```

Il ne faut pas utiliser un `else if`, car on mettrait en double l'instruction `print_string "c'est moi ou MOI"` :

```
let prenom = read_line() in
if prenom = "arthur" then
   print_string "c'est moi ou MOI";
else if prenom = "ARTHUR" then
   print_string "c'est moi ou MOI";
else
   print_string "quelqu'un d'autre"
```

5.2.3 NON logique

▶ On peut prendre le contraire d'une expression booléenne avec l'opérateur NON. En Caml, c'est la fonction prédéfinie `not` qui permet de faire cela. Ce mot se place devant l'expression booléenne dont on veut prendre la négation.

Ainsi `not true` vaut `false` et `not false` vaut `true`. On aurait pu définir cette fonction à la main en testant le paramètre `b` : si `b` est vrai, alors on renvoie faux, sinon c'est que `b` est faux, et on renvoie donc vrai. Cela s'écrit :

```
let not b =
   if b
      then false
      else true
   in
```

Une autre manière d'écrire un code équivalent, tester si `b` est égal à faux :

```
let not b =
   (b = false) in
```

▶ Pour tester si un entier est impair, on peut donc tester s'il n'est pas pair :

```
let n = read_int() in
if not (n mod 2 = 0)
   then print_string "impair"
   else print_string "pair";
```

Deuxième exemple, supposons que l'on ait déjà déclaré la fonction `est_pair` :

```
let est_pair n =
   (n mod 2 = 0) in
```

alors à ce moment, on peut construire la fonction `est_impair` en utilisant `est_pair` :

```
let est_impair n =
   not (est_pair n) in
```

5.2.4 Résumé

▶ Résumé des opérateurs de calcul sur les expressions booléennes :

- `(expr1 && expr2)` est vraie uniquement lorsque les deux expressions sont vraies.
- `(expr1 || expr2)` est vraie dès qu'une des deux expressions est vraie.
- `(not expr1)` est vraie lorsque l'expression est fausse, et fausse lorsque l'expression est vraie.

5.2.5 Exercices

★ Écrire une fonction qui prend trois valeurs de type quelconque en paramètres et détermine si elles sont toutes les trois égales.

– –

```
let trois_egaux x y z =
   (x = y) && (y = z) in
```

Remarque : il n'y a pas besoin de tester si `x = z`, puisque c'est automatiquement vérifié lorsque `x = y` et `y = z`.

On peut tester avec :

```
print_string (string_of_bool (trois_egaux 3 3 2));
print_string (string_of_bool (trois_egaux 3 3 3));
```

★ Écrire une fonction nommée `ou_exclusif` qui prend en paramètre deux booléens et qui renvoie vrai si l'un seul des deux est `true`. Ainsi vous devez avoir :

```
ou_exclusif true true     --> false
ou_exclusif true false    --> true
ou_exclusif false true    --> true
ou_exclusif false false   --> false
```

_ _

On pourra tester la fonction avec le code suivant, qui devra normalement afficher false, true, true, et false.

```
let ou_exclusif b1 b2 =
   ... in

let print_bool b =
   print_string (string_of_bool b);
   print_newline();
   in

print_bool (ou_exclusif true true);
print_bool (ou_exclusif true false);
print_bool (ou_exclusif false true);
print_bool (ou_exclusif false false);
```

Il y a pleins de façon de code la fonction `ou_exclusif`. En voici quelques-unes.

a) Le premier paramètre vrai mais pas le second, ou alors le second est vrai mais pas le premier :

```
let ou_exclusif b1 b2 =
   (b1 && (not b2)) || ((not b1) && b2) in
```

b) L'un ou l'autre est vrai, mais pas les deux à la fois :

```
let ou_exclusif b1 b2 =
   (b1 || b2) && (not (b1 && b2)) in
```

c) Négation de "les deux booléens sont égaux", c'est-à-dire qu'ils ne doivent pas être vrais en même temps, ni faux en même temps :

```
let ou_exclusif b1 b2 =
   (not (b1 = b2)) in
```

d) Ou de manière équivalente, les deux booléens sont différents :

```
let ou_exclusif b1 b2 =
   (b1 <> b2) in
```

Cette dernière solution est clairement la plus élégante.

★ Les années bissextiles ont lieux toutes les années dont la valeur est multiple de 4. Ainsi 1984, 1988, 1992, et 2004 sont bissextiles. Mais par exception, les années centenaires ne sont pas bissextiles. Ainsi 1800 et 1900 ne le sont pas. De plus, complication supplémentaire, les années multiples de 400 le sont ! Pouvez-vous écrire une fonction `est_bissextile` qui teste si une année est bissextile ?

L'année doit être multiple de 4 mais pas de 100, ou alors multiple de 400.

```
let est_bissextile an =
   ( (an mod 4 = 0) && (an mod 100 <> 0) ) || (an mod 400 = 0) in
```

Pour tester :

```
let an = read_int() in
let b = est_bissextile an in
print_string (string_of_bool b);
```

Autre façon de voir la chose : l'année doit être multiple de 4 dans tous les cas, et pour être bissextile, il faut qu'elle ne soit pas multiple de 100. ou alors qu'elle soit multiple de 400. Ce qui se traduit par la condition :

```
(an mod 4 = 0) && ( (an mod 100 <> 0) || (an mod 400 = 0) ) in
```

► Remarque : voici la fonction `est_bissextile` écrite en n'utilisant aucun opérateur booléen, explicitée avec des `if`.

```
let est_bissextile an =
   if (an mod 4) = 0 then
      begin
      if (an mod 100) = 0 then
         begin
         if (an mod 400) = 0
            then true
            else false
         end
      else
         false
      end
   else
      false
   in
```

Conclusion : les expressions booléennes permettent de simplifier grandement le code des programmes.

5.3 Exercices

5.3.1 Tarifs

★ Écrire une fonction `tarif_selon_age` qui prend en paramètre un âge, et qui donne le tarif sous forme de chaîne selon la règle suivante :

- "gratuit" jusqu'à 7 ans inclus,
- "tarif jeune" entre 8 et 25 ans inclus,
- "plein tarif" entre 26 et 65 ans inclus,

- "tarif vermeil" pour les plus de 66 ans.

_ _

Astuce : il n'y a pas besoin d'utiliser des ET, si on fait les tests dans le bon ordre avec des `else if`.

```
let tarif_selon_age age =
   if age <= 7 then
      "gratuit"
   else if age <= 25 then
      "tarif jeune"
   else if age <= 65 then
      "plein tarif"
   else
      "tarif vermeil"
   in
```

5.3.2 Recherche dans un tableau

★ Écrire une fonction `tableau_contient` qui en paramètre un tableau d'entiers `tab` d'une part, une valeur entière `clef` d'autre part, et qui renvoie un booléen pour savoir si au moins une des cases du tableau `tab` contient la valeur `clef`. Pouvez-vous donner le type de cette fonction ?

Attention : ne cherchez pas à interrompre la boucle `for` si vous rencontrez la valeur `clef`. Il n'est pas possible de quitter une boucle `for` avant d'arriver à la valeur finale du compteur. On verra plus tard une autre structure de boucle qui permet d'avoir un contrôle plus précis. Pour l'instant, on doit se contenter d'une boucle du type `for i = 0 to pred taille do ...` .

_ _

La fonction marche pour tous les tableaux : `'a array`. Pour que la comparaison entre les éléments du tableau et la clé ait un sens, la clé doit être de type `'a`. La fonction renvoie finalement un booléen, donc elle est de type `" 'a array -> 'a -> bool "`.

Pour tester votre fonction, vérifiez que ce code affiche `true` puis `false` :

```
let tab = [| 4; 9; -3; 5; 7 |] in
print_string (string_of_bool (contient tab 5));
print_string (string_of_bool (contient tab 3));
```

L'idée est d'utiliser une référence sur un booléen (type `bool ref`) nommée `trouve`. Cette référence est initialement à `false`, et devient `true` dès qu'on a trouvé au moins une fois la valeur que l'on cherche. Une boucle parcourt les éléments du tableau. Si on trouve la valeur voulue, on affecte `true` à la référence. Que renvoie-t-on à la fin ? Si on a trouvé `clef`, alors `!trouve` faut `true`, sinon il vaut `false`. On retournera donc la valeur de `!trouve`.

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      if tab.(i) = clef
         then trouve := true;
   done;
   !trouve in
```

Lorsqu'on a écrit la fonction permettant de récupérer le plus grand élément d'un tableau d'entiers, on avait vu qu'on pouvait mettre comme corps de la boucle l'instruction `maximum := max !maximum tab.(i)`. De la même manière, on peut dire ici qu'on a trouvé la valeur soit parce qu'on l'a déjà trouvé avant, soit parce qu'on vient de la trouver. On réunit donc ces deux cas avec un OU.

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      trouve := !trouve || (tab.(i) = clef);
   done;
   !trouve in
```

Cette seconde méthode permet de gagner une ligne, mais est peut-être un peu plus compliquée. Utilisez donc la méthode que vous préférez.

5.3.3 Tous positifs dans un tableau

★ Écrire une fonction `tableau_tous_positifs` qui prend en paramètre un tableau d'entiers, et qui renvoie vrai si toutes les valeurs contenues dans le tableau sont positives ou nulles.

Attention : on utilisera, comme dans l'exercice précédent, une boucle : `for i = 0 to pred taille do`.

_ _

Première méthode : une référence retient si tous les nombres déjà lu sont positifs. Cette référence est initialisée à `true`. Si jamais on trouve un nombre négatif, on passe la référence à `false`.

```
let tableau_tous_positifs tab =
   let tous_positifs = ref true in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      if tab.(i) < 0
         then tous_positifs := false;
   done;
   !tous_positifs in
```

Variante pour le corps de la boucle : `tous_positifs := !tous_positifs && (tab.(i) >= 0);`

Seconde méthode, une référence nommée `trouve` retient si au moins un nombre négatif a été trouvé. Le code ressemble alors beaucoup à celui de la recherche d'une valeur donnée dans le tableau : à la fin, on renvoie vrai si aucune valeur négative n'a été trouvée, c'est-à-dire qu'on retourne la négation du contenu de la référence : `not !trouve`.

```
let tableau_tous_positifs tab =
   let trouve = ref false in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      trouve := !trouve || (tab.(i) < 0);
   done;
   not !trouve in
```

5.4 Nombres aléatoires

5.4.1 Génération

▶ Dans cette partie, on va voir comment obtenir dans un programme un nombre aléatoire. Dans notre premier exemple, on va simuler un dé : il y a 6 valeurs possibles correspondant aux six faces du dé, et chaque face a autant de chances d'apparaître que les autres.

Pour ce faire, on utilise la fonction `Random.int`. Le mot `Random` veut dire aléatoire, et c'est le nom du module qui regroupe les fonctions de hasard. `Random.int` prend en paramètre un entier, qui représente le nombre de valeurs possibles parmi lesquelles on veut en piocher une.

`Random.int k` donne aléatoirement un entier compris entre 0 et `k-1`.

Pour notre dé qui a 6 faces, on fera donc `Random.int 6`. Le résultat de cette fonction est un entier qui vaut 0, 1, 2, 3, 4, ou 5. Remarquez que l'on a commencé à zéro, exactement comme pour les indices des tableaux. Pour obtenir une valeur au hasard entre 1 et 6, il suffit de rajouter 1 à ce nombre aléatoire. Voici notre dé :

```
let de = (Random.int 6) + 1 in
print_int de;
```

Testez ce programme plusieurs fois. Vous verrez qu'il affiche toujours la même valeur. C'est assez surprenant pour une fonction aléatoire ! Pour comprendre ce qu'il se passe, il faut connaître l'idée générale du fonctionnement des fonctions `Random`.

▶ Un ordinateur est surtout conçu pour faire des travaux répétitifs, toujours de la même manière. Si on lui demande de trouver un nombre au hasard, comme il appliquera toujours la même méthode de recherche, il trouvera toujours le même nombre !

Pour contourner ce problème, on fait retenir à l'ordinateur le dernier nombre aléatoire qu'il a trouvé. Et on lui donne une méthode de recherche telle que le résultat dépende du nombre aléatoire trouvé précédemment. Une fois un nombre aléatoire trouvé, on le retient, pour pouvoir construire le prochain.

Concrètement, une fois que le programme a choisi un premier nombre, il va l'utiliser pour générer le deuxième nombre, puis utiliser ce deuxième nombre pour fabriquer le troisième, et ainsi de suite. Si l'on exécute le même programme deux fois, et que les deux fois il choisi le même premier nombre, alors dans les deux exécutions la séquence qu'il va générer sera la même. Pour illustrer cela, regardez le programme suivant qui affiche 20 nombres aléatoires successifs. La séquence affichée est la même d'une exécution à l'autre.

```
for i = 1 to 20 do
   let r = Random.int 10 in
   print_int r;
   print_string " ";
done;
```

▶ Tout le problème est de choisir un nombre de départ qui soit différent entre chaque exécution. La contrainte étant qu'entre deux exécutions, le programme n'a pas le droit de stocker des données, et en particulier il ne peut pas retenir le nombre auquel il s'était arrêté. Alors comment faire pour que le programme puisse trouver un premier nombre aléatoire, afin que la séquence change à chaque exécution du programme ?

Une solution consiste par exemple à utiliser l'horloge de l'ordinateur : d'une exécution à l'autre du programme, l'heure ne sera pas exactement la même. Plutôt que de regarder nous-même l'horloge du système, on va demander au programme de se débrouiller tout seul pour trouver un point de départ à la séquence qui soit nouveau à chaque

exécution du programme. Pour ce faire, on utilise l'instruction `Random.self_init()`, qui veut dire "initialisation automatique du générateur de nombres aléatoires". Il n'y a pas besoin d'appeler cette fonction plus d'une fois.

L'instruction `Random.self_init()` doit être exécutée une fois pour toutes avant de pouvoir appeler le générateur de nombres aléatoires.

Avec cela, notre dé change à chaque fois :

```
Random.self_init();
let de = (Random.int 6) + 1 in
print_int de;
```

De même, la séquence de chiffres n'est plus la même à chaque fois :

```
Random.self_init();
for i = 1 to 20 do
   let r = Random.int 10 in
   print_int r;
   print_string " ";
done;
```

Remarque : on peut se demander pourquoi le `Random.self_init()` n'est pas ajouté automatiquement par le compilateur. La réponse est très simple : on ne veut pas ralentir les programmes qui n'ont pas besoin qu'un `self_init()` soit exécuté.

▶ Il est aussi possible de demander un nombre aléatoire réel.

La fonction `Random.float x` retourne un nombre aléatoire de type `float` compris entre zéro (inclus) et `x` (exclus).

Ainsi, `Random.float 2.3` donne une valeur au hasard entre 0.0 et 2.3. Par exemple :

```
Random.self_init();
let x = Random.float 2.3 in
print_float x;
```

peut afficher `0.323020527156`, si c'est ce nombre aléatoire qui a été choisi.

▶ Deux remarques. Premièrement, on verra en exercice comment choisir un booléen au hasard. Deuxièmement, il n'y a pas de fonctions permettant de piocher une lettre au hasard, on verra comment faire cela dans un autre chapitre, car cela demande d'en connaître plus sur les caractères.

5.4.2 Intervalle donné

★ Ecrivez un programme qui affiche un nombre aléatoire entre 4 et 12 inclus.

– –

Il y a 9 valeurs possibles. Pour le vérifiez, vous pouvez compter sur vos doigts, ou bien mieux, calculer `(12 - 4 + 1)`. Avoir 9 choix possibles nous indique qu'il faut appeler `Random.int 9`, ce qui va nous donner une valeur entre 0 et 8 inclus. Il n'y aura plus qu'à ajouter 4 à ce nombre pour obtenir une valeur entre 4 et 12 inclus.

```
Random.self_init();
let x = (Random.int 9) + 4 in
print_int x;
```

★ Ecrivez un programme qui affiche un nombre aléatoire entre -100 et +100 exclus, (c'est-à-dire entre -99 et +99 inclus).

_ _

Il y a `99 - (-99) + 1 = 199` valeurs possibles (on peut aussi compter : 99 strictement en-dessous de zéro, 99 strictement au-dessus de zéro, plus 1 pour le zéro). `Random.int 199` nous donne une valeur entre 0 et 198 inclus. Il n'y a qu'a lui retrancher 99 pour obtenir un entier dans l'intervalle souhaité.

```
Random.self_init();
let x = (Random.int 199) - 99 in
print_int x;
```

★ Écrire une fonction `rand_float_entre` qui prend deux paramètres flottants `x` et `y`, en supposant que `x < y`, et qui donne un nombre réel aléatoire compris entre `x` et `y` (inclus ou exclus, cela ne change quasiment rien).

_ _

L'idée est de chercher un nombre aléatoire dans l'intervalle 0 à `y-x`, pour ensuite ajouter `x` à cette valeur. Voici le code complet, avec un petit test.

```
Random.self_init();

let rand_float_entre x y =
   (Random.float (y -. x)) +. x in

print_float (rand_float_entre (-3.4) 1.2);
```

Remarque : le `Random.float` n'est exécuté que lorsque la fonction est appelée. Il est donc tout à fait possible de placer le `Random.self_init()` après la déclaration de la fonction :

```
let rand_float_entre x y =
   (Random.float (y -. x)) +. x in

Random.self_init();
print_float (rand_float_entre (-3.4) 1.2);
```

5.4.3 Probabilité donnée

★ En utilisant la fonction `Random.int`, écrivez une fonction `random_bool` de type `unit -> bool` qui retourne un booléen au hasard. C'est-à-dire retourner `true` ou `false` avec une probabilité de une chance sur deux.

_ _

Il y a deux valeurs possibles, on utilise donc un `Random.int 2`. Cette fonction renvoie soit 0, soit 1. On a qu'a dire ensuite qu'on renvoie `true` si c'est 0, et `false` si c'est 1. Ce qu'il faut éviter de faire :

```
let rand_bool () =
   if (Random.int 2) = 0
      then true
      else false
   in
```

La bonne solution, plus courte :

```
let rand_bool () =
   ((Random.int 2) = 0) in
```

★ Faites la même chose que l'exercice précédent, c'est-à-dire écrivez une fonction `random_bool` mais cette fois en utilisant `Random.float`.

_ _

Avec `Random.float`, on peut utiliser un intervalle quelconque. On renverra `true` si la valeur tombe dans la première moitié de l'intervalle, et `false` sinon. Il n'y pas besoin de se soucier de la valeur pile au milieu, car on ne tombe virtuellement jamais juste dessus. On a le choix de l'intervalle utilisé.

En utilisant un intervalle de taille `1.0`, le milieu se situe à `0.5`, c'est-à-dire que l'on retourne vrai si le nombre aléatoire est inférieur à `0.5` et faux sinon :

```
let rand_bool () =
   ((Random.float 1.0) < 0.5) in
```

En utilisant un intervalle de taille `2.0`, le milieu se situe à `1.0`, c'est-à-dire que l'on retourne vrai si le nombre aléatoire est inférieur à `1.0` et faux sinon :

```
let rand_bool () =
   ((Random.float 2.0) < 1.0) in
```

► Remarque : la fonction `Random.bool()` est en fait prédéfinie, et on peut donc l'utiliser directement. L'exercice n'était pas pour autant inutile, car il va permettre de traiter plus facilement les deux suivants.

`Random.bool()` a une chance sur deux de renvoyer `true`, et une chance sur deux de renvoyer `false`.

★ Ecrivez la fonction `proba_bool` qui prend en paramètre un entier `k`, et qui renvoie `true` avec une probabilité de une chance sur `k`. Cela veut dire que si on appelle `k` fois cette fonction, alors, en moyenne, elle ne renverra vrai qu'une seule fois. On peut aussi dire que la fonction renvoie en moyenne vrai tous les `k` appels. Testez ensuite 200 fois cette fonction pour `k = 4`, en affichant un 1 à chaque fois que la fonction renvoie vrai, et un 0 lorsqu'elle renvoie faux. On ne mettra ni espace ni retour à la ligne.

Indication : si vous ne voyez vraiment pas du tout ce que l'énoncé veut dire, voici ce qu'il faut faire. Prenez une valeur au hasard parmi `k` entiers, et renvoyez vrai lorsque cette valeur est égale à zéro, faux sinon.

_ _

Voici la fonction :

```
let proba_bool k =
   ((Random.int k) = 0) in
```

Et de quoi la tester, avec `k = 4` :

```
Random.self_init();

let proba_bool k =
   ((Random.int k) = 0) in

for i = 1 to 200 do
```

```
    if proba_bool 4
       then print_int 1
       else print_int 0;
done;
```

Pouvez-vous voir qu'il y a environ trois 0 pour chaque 1 affiché ? Remarquez également qu'on peut parfois avoir de très longues suites de 0 (par exemple une douzaine).

5.4.4 Dessins aléatoires

★ Ecrivez une fonction `rectangle_hasard` qui prend en paramètre deux entiers `nb_lignes` et `nb_colonnes`, et qui affiche un rectangle de ces dimensions, contenant des dièses `'#'` et des point `'.'`. Ces caractères doivent être choisis au hasard, avec une probabilité de une chance sur deux. Voici un exemple pour les paramètres 6 lignes et 15 colonnes :

```
#..#.#...#..#.#
#..##.#..##.###
...##.#.##..###
.##.##....#.##.
#####.#########
.....#....#.###
```

Modifiez ensuite votre code pour affichez un rectangle contenant des `'#'` avec une probabilité 1 chance sur `k`, où `k` est un troisième paramètre de la fonction. Testez alors votre fonction pour diverses valeurs de `k`, y compris des valeurs assez grandes sur de grands rectangles. Voici un exemple pour les paramètres 3 lignes et 50 colonnes, avec `k=10` :

```
.....#......#.................#....#..............
.....................#..#.....#........#....#.....
........#.....#...#....#.................#.......#
```

_ _

Pour le hasard, on peut utiliser au choix `Random.bool`, `Random.int` ou `Random.float`. Le premier est le plus simple. Pour l'affichage, on peut utiliser `print_char` ou `print_string`. `print_char` est cependant plus logique puisqu'on affiche des caractères un par un.

```
let rectangle_hasard nb_lignes nb_colonnes =
    for ligne = 1 to nb_lignes do
       for colonne = 1 to nb_colonnes do
          if Random.bool()
             then print_char '#'
             else print_char '.';
       done;
       print_newline();
    done;
    in
```

Pour tester, il ne faut pas oublier le `Random.self_init()` avant l'appel à `rectangle_hasard`.

```
Random.self_init();
rectangle_hasard 15 25;
```

Pour avoir une probabilité de 1 sur `k`, il suffit de remplacer la ligne `if Random.bool()` par `if (Random.int k) = 0`, et de rajouter `k` comme paramètre. Exemple pour `k = 8`.

```
let rectangle_hasard nb_lignes nb_colonnes k =
   for ligne = 1 to nb_lignes do
      for colonne = 1 to nb_colonnes do
         if (Random.int k) = 0
            then print_char '#'
            else print_char '.';
      done;
      print_newline();
   done;
   in

Random.self_init();
rectangle_hasard 15 50 8;
```

Remarque : on aurait pu aussi utiliser la fonction `proba_bool` en faisant `if proba_bool k`.

Programming is easy, isn't it ?

Chapitre 6

while, rec

6.1 La boucle while

6.1.1 Structure

▶ La boucle `for` permet de répéter du code un certain nombre de fois. Mais ce nombre de répétition doit obligatoirement être connu au moment où l'on commence la boucle, lorsqu'on précise la valeur initiale et la valeur finale du compteur. Après cela, il n'y a plus aucun moyen d'interrompre la boucle, c'est-à-dire de l'arrêter avant que le compteur soit arrivé à sa valeur finale.

Il existe donc une autre boucle, la **boucle while** qui permet de contrôler comme on veut le nombre d'exécution. En effet, avant chaque exécution du corps de cette boucle, on peut choisir si on veut exécuter ce corps de la boucle ou bien si préfère s'arrêter et passer à la suite. Concrètement, la boucle s'effectue tant qu'une certaine condition est réalisée, c'est-à-dire est évaluée comme vraie.

Voici le schéma de la boucle `while` :

```
while (...condition...) do
   (...corps de la boucle while...)
done;
```

Explications :

- Le mot clé `while` se traduit `tant que`.
- La condition est une expression booléenne.
- Le corps d'une boucle `while` est analogue au corps d'une boucle `for`. En particulier, une déclaration effectuée à l'intérieur du corps d'une boucle a une portée limitée à ce corps.

Remarque sur les extrêmes :

- Si la condition est fausse la première fois qu'on l'évalue, le corps de la boucle ne sera pas exécuté du tout.
- Si la condition est toujours vraie, alors le programme ne s'arrêtera pas. On appelle cela une **boucle infinie**.

Comment la condition peut-elle être vraie au début, et devenir faux au bout d'un certain moment ? Il y a deux possibilités, on va donc donner deux exemples.

► La première possibilité est que la condition dépende d'une valeur donnée par l'utilisateur. Le programme que l'on va étudier simule un coffre fort. Ce programme demande à l'utilisateur un entier. Si cette combinaison est le code secret, alors le coffre s'ouvre, mais sinon, le programme indique à l'utilisateur qu'il s'est trompé et qu'il doit recommencer.

Au début, le programme attend que l'utilisateur tape un nombre. Lorsque le bon code est trouvé, il affiche `"Coffre ouvert !"`. A chaque fois que l'utilisateur se trompe, il affiche `"Code non valide !"`, puis recommence depuis le début. Dans cet exemple, le code secret est 36. La condition de la boucle s'exprime donc par l'expression `(read_int() <> 36)`. Si l'utilisateur donne 36, cette expression est fausse et la boucle s'arrête. Dans les autres cas, cette expression est vraie, et le coffre reste fermé et le programme recommence.

```
while read_int() <> 36 do
   print_string "Code non valide !\n";
done;
print_string "Coffre ouvert !\n";
```

► La seconde possibilité pour que la condition de la boucle change au bout d'un moment est que cette condition dépende de la valeur d'une référence. Dans ce cas, la référence peut être modifiée dans le corps de la boucle.

L'exemple suivant initialise une référence `i` à 0, et tant que le contenu de `i` est inférieur à 20, le corps de la boucle se chargera d'afficher ce contenu, puis de l'augmenter de 1 (c'est-à-dire incrémenter `i`). Voici le code :

```
let i = ref 0 in
while !i <= 20 do
   print_int !i;
   print_string " ";
   incr i;
done;
print_string "c'est fini";
```

Remarque : on pourrait remplacer `while !i <= 20 do` par `while !i <> 20 do` puisqu'on sait que comme les valeurs de `!i` augmentent, `!i` finira par atteindre 20. On va comprendre dans l'exercice suivant pourquoi il ne faut pas le faire.

★ Le code précédent montre qu'on effectue un travail équivalent à celui d'une boucle `for` à l'aide d'une boucle `while`. Ecrivez donc une fonction nommée `affiche_entiers` qui prend en paramètre deux entiers nommés `debut` et `fin`, et qui affiche à l'aide d'une boucle `while` tous les entier compris entre ces deux valeurs (incluses). Les entiers doivent être séparés par des espaces.

Remarque : avec une boucle `for`, si `debut` est strictement supérieur à `fin`, le corps de la boucle n'est pas exécuté. Il faut donc que la fonction `affiche_entiers` n'affiche rien du tout dans un tel cas.

– –

La fonction, avec un petit test.

```
let affiche_entiers debut fin =
   let i = ref debut in
   while !i <= fin do
      print_int !i;
      print_string " ";
      incr i;
   done;
```

```
   in

affiche_entiers 23 54;
```

Lorsque `debut > fin`, la référence `i` est initialisée à `debut`, et la condition `!i <= fin` est fausse dès la première fois qu'elle est évaluée. Ainsi la fonction n'affichera rien.

En revanche, ce n'aurait pas été le cas si la condition était `while !i <> fin do`. Dans ce cas, cette condition serait toujours vrai, et on a une boucle infinie : tous les entiers à partir de `debut` sont affichés les uns après les autres, et le programme pas tout seul. Pour pouvoir mieux voir ce qui est affiché, on utilise `print_newline()` à la place de `print_string " "` :

```
let affiche_entiers debut fin =
   let i = ref debut in
   while !i <> fin do
      print_int !i;
      print_newline();
      incr i;
   done;
   in

affiche_entiers 15 3;
```

6.1.2 Recherche dans un tableau

► Lorsqu'on a écrit la fonction permettant de rechercher une valeur donnée dans un tableau, on a remarqué qu'une boucle `for` n'était pas très efficace. En effet, même si on trouve la valeur que l'on cherche tout au début du tableau, on est obligé d'aller jusqu'au bout de la boucle. Ce qui est inutile puisqu'on sait déjà qu'on veut renvoyer vrai.

La boucle `while` va nous permettre d'effectuer une recherche dans un tableau, en s'arrêtant soit parce qu'on est arrivé au bout du tableau, soit parce qu'on a déjà trouvé la valeur que l'on cherchait. Pour écrire la nouvelle fonction de recherche, on va procéder en deux étapes. D'abord transformer la boucle `for` en une boucle `while` équivalente, c'est-à-dire en continuant d'aller jusqu'au bout du tableau. Ensuite, on modifiera la condition de la boucle `while` afin de s'arrêter dès qu'on a trouvé la valeur que l'on cherchait.

★ Première étape : modifiez la fonction suivante pour qu'elle fasse exactement la même chose, mais avec une boucle `while`

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   for i = 0 to pred taille do
      if tab.(i) = clef
         then trouve := true;
   done;
   !trouve in
```

- -

C'est exactement le même principe que pour afficher les entiers consécutifs. On déclare une référence `i`, initialement à 0. La condition d'arrêt est donc `(!i <= pred taille)`, ce qui peut également s'écrire `!i < taille`. Il faut modifier `chaine.[i]` en `chaine.[!i]` puisque `i` est maintenant une référence. Et il ne faut pas oublier d'ajouter `incr i` à la fin de la boucle, sinon `i` resterait à zéro, et on aurait une boucle infinie. On a donc :

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   let i = ref 0 in
   while !i < taille do
      if tab.(!i) = clef
         then trouve := true;
      incr i;
   done;
   !trouve in
```

★ Deuxième étape : modifiez la fonction précédente pour que la boucle `while` s'arrête dès que la `clef` a été trouvé.

_ _

Il s'agit de modifier la condition de la boucle `while`, pour qu'elle s'arrête soit lorsqu'on arrive au bout du tableau, soit lorsqu'on a trouvé la valeur `clef`. La boucle continue à s'exécuter tant que `(!i < taille)` et que `!trouve = false`. D'où la fonction :

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   let i = ref 0 in
   while (!i < taille) && (!trouve = false) do
      if tab.(!i) = clef
         then trouve := true;
      incr i;
   done;
   !trouve in
```

Remarque : l'expression `(!trouve = false)` est équivalente à `(not !trouve)`.

▶ Comment s'assurer que la boucle s'arrête bien dès qu'on trouve quelque la valeur `clef` ? Il suffit d'afficher les valeurs de `!i` pour lesquels la boucle est effectuée. Ainsi :

```
let tableau_contient tab clef =
   let trouve = ref false in
   let taille = Array.length tab in
   let i = ref 0 in
   while (!i < taille) && (!trouve = false) do
      print_int !i;
      if tab.(!i) = clef
         then trouve := true;
      incr i;
   done;
   !trouve in

let print_bool b =
   print_string (string_of_bool b);
   print_newline();
   in

let t = [| 3; 4; 7; -4; 4; 2 |] in
print_bool (tableau_contient t 4);
```

```
print_bool (tableau_contient t (-4));
print_bool (tableau_contient t 8);
```

Affiche :

```
01true
0123true
012345false
```

6.1.3 Lancés de dé

▶ Le but de cette partie est d'étudier le nombre de fois qu'il faut en moyenne lancer un dé avant d'obtenir un 6.

★ Ecrivez un programme qui lance un dé au hasard jusqu'à obtenir un 6, et qui affiche à la fin le nombre d'essais qui ont été nécessaires. Affichez à chaque lancé la valeur du dé.

– –

On va avoir besoin de deux références. La première, nommée `nb_essais`, compte le nombre de tentatives déjà effectuées. `nb_essais` est initialisé à 0. La seconde, nommée `continue`, est un booléen qui est vrai s'il faut continuer à lancer le dé, et qui devient faux lorsqu'on a obtenu un 6. `continue` est initialisé à `true`.

La boucle `while` s'exécute tant que `!continue` est vrai. A chaque exécution du corps, on dit aussi à chaque **itération de la boucle**, il faut lancer un dé, afficher sa valeur, incrémenter `nb_essais`, et enfin affecter à `continue` la valeur `false` si on a obtenu un 6. A là fin, on affiche la valeur de `!nb_essais`. Au fait, n'oubliez pas le `Random.self_init()` tout au début.

```
Random.self_init();
let nb_essais = ref 0 in
let continue = ref true in
while !continue do
   let de = (Random.int 6) + 1 in
   print_int de;
   incr nb_essais;
   if de = 6
      then continue := false;
done;
print_newline();
print_int !nb_essais;
```

Première remarque : on peut mettre l'instruction `incr nb_essais` n'importe où dans le corps de la boucle, cela ne change rien.

Seconde remarque : on peut remplacer le test avec `if` par simplement `continue := (de <> 6)`, ce qui affecte `true` lorsque 6 a été tiré, et `false` sinon.

▶ Testez plusieurs fois le programme pour voir un peu combien d'étapes sont nécessaires. Une fois sur 6, on tombe directement sur un 6 :

```
6
1
```

Parfois c'est le contraire : il faut beaucoup d'essais avant d'arriver à obtenir un 6. Certains résultats peuvent surprendre. Voici un exemple de véritable malchance :

```
14111252243331534215533153311235414 56
37
```

★ Afin de pouvoir faire plus de tests, écrivez une fonction qui renvoie le nombre de lancés qui ont été nécessaires pour obtenir un 6. Cette fonction ne doit plus rien afficher. Testez alors cette fonction avec une boucle `for` pour afficher une centaine de résultats.

Ensuite, calculez la valeur moyenne de tous ces résultats, afin d'avoir une estimation du nombre de lancés nécessaires en moyenne. Attention, une moyenne est toujours de type `float`.

_ _

La fonction :

```
let tentatives_pour_6 () =
   let nb_essais = ref 0 in
   let continue = ref true in
   while !continue do
      let de = (Random.int 6) + 1 in
      incr nb_essais;
      continue := (de <> 6);
   done;
   !nb_essais in
```

Remarque : on peut écrire une fonction équivalente en moins de lignes, comme cela :

```
let tentatives_pour_6 () =
   let nb_essais = ref 1 in
   while ((Random.int 6) + 1) <> 6 do
      incr nb_essais;
   done;
   !nb_essais in
```

Pourquoi n'a-t-on pas fait ça depuis le début ? Parce qu'avec cette méthode, on ne peut plus afficher la valeur du dé, vu qu'elle n'est pas nommée.

Pour afficher les résultats, on fait donc :

```
Random.self_init();

let tentatives_pour_6 () =
   let nb_essais = ref 1 in
   while ((Random.int 6) + 1) <> 6 do
      incr nb_essais;
   done;
   !nb_essais in

for i = 1 to 100 do
   let nb_essais = tentatives_pour_6 () in
   print_int nb_essais;
   print_string " ";
done;
```

Pour calculer la moyenne des résultats, on en fait la somme, et on divise ensuite par le nombre de valeurs. On pose `nb_valeurs` le nombre de fois que l'on veut appeler la fonction `tentatives_pour_6`, qui vaudra pour commencer 100. Remplacez donc la boucle `for` du code précédent par :

```
let nb_valeurs = 100 in
let somme = ref 0 in
for i = 1 to nb_valeurs do
   let nb_essais = tentatives_pour_6 () in
   somme := !somme + nb_essais;
done;
let moyenne = (float_of_int !somme) /. (float_of_int nb_valeurs) in
print_float moyenne;
```

► Après plusieurs essais, vous remarquerez que la valeur tourne autour de 6. C'est normal : vu qu'on a une chance sur 6 de tomber sur la face six, il faut en moyenne 6 essais avant de tomber sur cette face.

Plus vous donnerez à `nb_valeurs` une grande valeur, et plus vous serez près de la valeur théorique 6. Testez ainsi avec `nb_valeurs = 1000` puis `nb_valeurs = 10000`.

6.2 Exercices

6.2.1 Suite de Syracuse

► Dans cette partie, on va étudier la remarquable suite de Syracuse.

★ Pour commencer, il faut définir la fonction de Syracuse. Cette fonction, nommée `syracuse`, prend en paramètre un entier nommé `p`. Si `p` est pair, elle renvoie `p / 2` (la moitié de p). Si `p` est impair, elle renvoie `3 * p + 1`.

_ _

```
let syracuse p =
   if p mod 2 = 0
      then p / 2
      else 3 * p + 1
   in
```

► Le principe de la suite de la Syracuse est le suivant. On part d'un entier quelconque, qu'on nomme `n`. On calcule alors `syracuse n`. Cela nous donne une nouvelle valeur. On applique alors la fonction `syracuse` à cette nouvelle valeur. Ce qui nous donne une troisième valeur. On continue ainsi jusqu'à arriver à 1 (ce qui est remarquable, c'est qu'on finit par y arriver !).

Prenons par exemple `n = 7`. Ce nombre est impair, on le multiplie par 3 et on ajoute 1, ce qui donne 22. Comme 22 est pair, `syracuse 22` renvoie 11. De 11, on va à `3*11+1`, soit 34. On divise par 2, 17. De 17 on va à 52. Puis à 26. Ensuite 13, puis 40, ensuite 20, et 10, puis 5. Après 16, puis 8, 4, 2, et enfin 1 !

La suite de Syracuse pour le nombre 7 est ainsi :

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Pourquoi s'arrête-t-on à 1 ? Car `syracuse 1` vaut 4, et que de 4 on redescend à 2 puis à 1. Donc une fois qu'on arrive à 1, on ne fait plus que tourner en rond.

Vous ne croyez pas qu'on arrive toujours à 1 ? Essayer un autre nombre : 6 par exemple.

```
6 3 10 5 16 8 4 2 1
```

★ Les calculs étant assez fastidieux à faire à la main, on va vite écrire un petit programme qui calcule pour nous la suite de Syracuse d'un nombre donné. Ecrivez donc la fonction `affiche_syracuse`, à l'aide d'une boucle `while` et de la fonction `syracuse`. Cette fonction prend en paramètre l'entier de départ, et affiche les étapes de la suite, jusqu'au 1. Il n'est pas nécessaire d'afficher l'entier de départ au début, puisqu'on vient de le taper.

– –

```
let syracuse p =
   if p mod 2 = 0
      then p / 2
      else 3 * p + 1
   in

let affiche_syracuse depart =
   let n = ref depart in
   while !n <> 1 do
      n := syracuse !n;
      print_int !n;
      print_string " ";
   done;
   in
```

★ En réalisant plusieurs tests, vous pouvez vous rendre compte que le nombre d'étapes nécessaires avant d'arriver à 1 est très variable selon l'entier de départ. Essayez donc d'obtenir, pour chaque valeur de départ compris entre 1 et 100, le nombre d'étapes nécessaires pour arriver à 1.

– –

On va déjà écrire la fonction `etapes_syracuses` qui prend en paramètre un entier `depart` et qui retourne le nombre de fois qu'il faut appliquer la fonction `syracuse` avant d'arriver à 1 :

```
let etapes_syracuses depart =
   let nb_etapes = ref 0 in
   let n = ref depart in
   while !n <> 1 do
      n := syracuse !n;
      incr nb_etapes;
   done;
   !nb_etapes in
```

Affichons maintenant les valeurs de cette fonction pour tous les entiers compris entre 1 et 100. Rajoutez après la définition des fonctions `Syracuse` et `etapes_Syracuses` le code :

```
for i = 1 to 100 do
   print_int i;
   print_string " -> ";
   print_int (etapes_suite i);
   print_newline();
done;
```

Quelques morceaux de l'affichage obtenu :

```
1 -> 0
2 -> 1
3 -> 7
...
27 -> 111
28 -> 18
...
73 -> 115
74 -> 22
75 -> 14
...
99 -> 25
100 -> 25
```

▶ La suite de Syracuse semble toujours aboutir à 1 au bout d'un nombre fini d'étapes. Personne n'a encore réussi à prouver mathématiquement qu'il en était ainsi pour tous les entiers de départ possibles. Mais cela a été vérifié pour tous les entiers qui ont pu être testé. C'est ce qu'on appelle une conjecture mathématique.

6.2.2 Le jeu "c'est plus, c'est moins"

★ Le but de cet exercice est de programmer le jeu "c'est plus, c'est moins". Dans ce jeu, l'ordinateur commence par choisir au hasard un nombre entier entre 0 et 99 inclus. Le but du joueur, l'utilisateur du programme, est de deviner ce nombre. Pour cela il donne un nombre, et le programme lui répond `"C'est plus"` si le nombre à deviner est plus grand que celui qui a été donné, ou `"C'est moins"` si le nombre à deviner est plus petit que celui qui a été donné. Lorsque le joueur devine le bon nombre, le programme affiche `"C'est gagné"`, et précise ensuite le nombre de tentatives qui ont été nécessaires pour gagner.

_ _

```
Random.self_init();
let cible = Random.int 100 in
let nb_essais = ref 0 in
let continue = ref true in

while !continue do
   incr nb_essais;
   print_string "Essayez un nombre : ";
   let essai = read_int() in
   if cible = essai then
      continue := false
   else if cible < essai then
      print_string "C'est moins\n"
   else
      print_string "C'est plus\n";
done;

print_string "C'est gagné !\n";
print_string "Nombre de coups : ";
print_int !nb_essais;
```

▶ Si vous jouez bien, vous devez gagner en 7 coups ou moins à chaque partie.

6.2.3 Course aux dés

★ Deux joueurs font la course aux dés. Ils partent tous deux avec leur pion de la case numéroté 0, et se dirigent vers la case 30 qui est l'arrivée. A chaque tour, les deux joueurs jouent ensembles et avancent leur pion du nombre de cases indiquées par le dé. Le premier joueur qui dépasse la case numéro 30 gagne la partie. Si les deux joueurs arrivent sur l'arrivée ensembles, ils sont déclarés ex-æquo.

Vous devez écrire un programme qui simule une partie, en affichant à chaque tour la position des deux joueurs. A la fin de la partie, affichez le résultat.

– –

On va voir deux approches du problème. Dans les deux cas, on utilise deux références pour retenir la position des joueurs, `joueur1` et `joueur2`, initialement à 0. Lorsque le contenu d'une ou bien des deux de ces références est supérieur ou égal à 30, la partie est finie.

▶ Première méthode : on joue tant que `(!joueur1 < 30) && (!joueur2 < 30)`. Dans le corps de la boucle, on effectue les deux lancés de dés, puis on affiche la position des joueurs. Après la boucle, on effectue un test pour analyser le résultat. Voici un tel programme :

```
Random.self_init();
let joueur1 = ref 0 in
let joueur2 = ref 0 in

while (!joueur1 < 30) && (!joueur2 < 30) do
   joueur1 := !joueur1 + (Random.int 6) + 1;
   joueur2 := !joueur2 + (Random.int 6) + 1;
   print_int !joueur1;
   print_string " - ";
   print_int !joueur2;
   print_newline();
done;

if (!joueur1 >= 30) && (!joueur2 >= 30) then
   print_string "Joueurs ex-aequo";
else if (!joueur1 >= 30) then
   print_string "Joueur 1 gagne";
else if (!joueur2 <= 30) then
   print_string "Joueur 2 gagne";
```

Première remarque : on peut aussi utiliser comme condition de boucle : `while (max joueur1 joueur2) < 30 do`. C'est un peu plus court, mais ça complique un peu la compréhension du code.

Seconde remarque : on peut utiliser un `else` pour la dernière condition, mais cela casse la symétrie du code, donc ce n'est pas forcément intéressant :

```
if (!joueur1 >= 30) && (!joueur2 >= 30) then
   print_string "Joueurs ex-aequo";
else if (!joueur1 >= 30) then
   print_string "Joueur 1 gagne";
else
   print_string "Joueur 2 gagne";
```

▶ Seconde méthode : on utilise une référence sur une chaîne pour retenir l'état du jeu. Cette référence nommée `status` peut contenir 4 valeurs : `"en cours"`, `"Joueur 1 gagne"`, `"Joueur 2 gagne"`, ou `Joueurs ex-aequo`.

Une fois que les joueurs ont avancé leur pion, on réalise un test pour mettre à jour l'état du jeu. Après la boucle, il n'y a plus qu'à afficher le contenu de `status`.

```
Random.self_init();
let joueur1 = ref 0 in
let joueur2 = ref 0 in
let status = ref "en cours" in

while !status = "en cours" do
   joueur1 := !joueur1 + (Random.int 6) + 1;
   joueur2 := !joueur2 + (Random.int 6) + 1;

   if (!joueur1 >= 30) && (!joueur2 >= 30) then
      status := "Joueurs ex-aequo";
   else if (!joueur1 >= 30) then
      status := "Joueur 1 gagne";
   else if (!joueur2 >= 30) then
      status := "Joueur 2 gagne";

   print_int !joueur1;
   print_string " - ";
   print_int !joueur2;
   print_newline();
done;

print_string !status;
```

Remarque : pour mettre à jour `status`, on pourrait aussi mettre dans le corps de la boucle la structure :

```
   if (!joueur1 >= 30)
      then status := "Joueur 1 gagne";
   if (!joueur2 >= 30)
      then status := "Joueur 2 gagne";
   if (!joueur1 >= 30) && (!joueur2 >= 30)
      then status := "Joueurs ex-aequo";
```

Dans ce cas, si les joueurs arrivent ensembles, `status` sera modifié à chacune des instructions, ce qui est moins clair que l'autre structure, qui différencie bien les cas avec des `else if`.

Remarque : on verra plus tard les **unions**, une autre méthode plus propre permettant de coder les différents états possibles d'un programme (ce que l'on fait ici avec la référence sur une chaîne `status`.

6.3 Fonctions récursives

6.3.1 Plans de robots

► On a expliqué les fonctions en prenant l'images de robots. Rappelons le principe : lorsque le robot est appelé, il ramasse les objets qui sont posés devant lui, il effectue des actions, puis il dépose éventuellement un objet avant de partir. On pourrait croire que pour chaque fonction, on a un robot qui dort dans un coin, et qu'à chaque fois que la fonction est appelée, on réveil le robot pour qu'il fasse son travail. Eh bien ce n'est pas tout à fait comme cela que ça se passe.

En fait, une fonction, c'est un manuel de fabrication pour un robot. Au début, aucun robot n'est présent. Les robots sont fabriqués dès qu'on a besoin d'eux, et jetés à la poubelle dès qu'on en a plus besoin. La fonction que le programmeur écrit correspond aux plans de constructions du robot. En d'autres termes, lorsqu'on appelle une fonction, un robot est construit, ce robot fait son travail, puis il est détruit. Ce n'est pas très écologique de gaspiller autant de robots, mais comme tout ce qu'on fait est entièrement virtuel, aucune pollution n'est engendrée.

► Pour faire son travail, un robot peut faire appel à d'autres robots. Par exemple, la fonction suivant affiche successivement ses deux paramètres qui sont de type `int` :

```
let print_2_int x y =
   print_int x;
   print_int y;
   in
```

Supposons que l'on écrive maintenant `print_2_int 9 5`. Détaillons ce qui se passe :

1. Un robot A est créé selon le modèle `print_2_int`. A est donc programmé pour ramasser deux objets et appeler `print_int` sur chacun d'entre eux.
2. Le robot A est ensuite mis en route. Il ramasse le 9 et le 5.
3. Ensuite, A appelle `print_int 9`. Pour cela, il pose le 9, et demande la construction d'un robot avec le plan de construction `print_int`.
   (a) Un robot B est crée selon le modèle `print_int`.
   (b) Le robot B ramasse l'entier 9, et l'affiche.
   (c) Le robot B a fini son travail, il s'auto-détruit.
4. Après, A appelle `print_int 4`. Pour cela, il pose le 4, et demande la construction d'un robot avec le plan de construction `print_int`.
   (a) Un robot C est crée selon le modèle `print_int`.
   (b) Le robot C ramasse l'entier 4, et l'affiche.
   (c) Le robot C a fini son travail, il s'auto-détruit.
5. Le robot A a fini son travail, il s'auto-détruit.

Il est important de noter qu'il n'y a aucune relation entre le robot B, qui affiche le 9, et le robot C, qui affiche le 4, si ce n'est qu'ils ont été fabriqué tous deux selon le même plan de fabrication, à savoir la fonction `print_int`.

6.3.2 Définition de récursivif

► Un robot peut donc appeler d'autres robots. Dans l'exemple qui précède, un robot programmé selon la fonction `print_2_int` appelait des robots programmés selon la fonction `print_int`.

Maintenant, il est possible qu'un robot, pour effectuer sont travail, fasse appel à un autre robot qui soit programmé de la même manière. Traduit en termes de fonctions, cela donne :

Une fonction récursive est une fonction qui s'appelle elle-même.

► Voyons tout de suite un exemple. On va fabriquer un robot qui ramasse un entier `n`, et qui va calculer la valeur de la factorielle de `n`, c'est on le rappelle le produit des `n` premiers entiers. On sait déjà coder ça avec des

boucles `for` ou `while`, mais cette fois on va faire sans. Pour cela, on va partir du principe que le produit des entiers compris entre 1 et `n` est égal à `n` fois le produit des entiers compris entre 1 et `n-1`.

Pour calculer factorielle de `n`, notre robot va donc procéder en deux étapes. D'abord il va faire appel à un robot assistant (un robot distinct, mais programmé pareil) pour calculer factorielle de `(n - 1)`. Ensuite il va multiplier le résultat obtenu par `n`, et retourner le résultat.

Que fait le robot assistant pour calculer factorielle de `(n - 1)` ? Il fait la même chose : il utilise encore un autre robot assistant, qui sera chargé de calculer la factorielle de `(n - 2)`, puis il multiplie le résultat par `(n - 1)`. Mais quand cela s'arrête-t-il ? Au bout d'un moment, il faudra calculer la factorielle de 1. Pour cela, il n'y a pas besoin d'utiliser de robot assistant, car la valeur du résultat est trivial : c'est 1.

▶ On peut maintenant écrire le plan de fabrication de notre robot, c'est-à-dire la fonction récursive `factorielle` :

```
let factorielle n =
   if n = 1
      then 1
      else n * (factorielle (n-1))
   in
```

- La fonction prend un entier `n` en argument.
- Si `n` vaut 1, alors on retourne 1.
- Sinon, on retourne `n` fois la factorielle de `n - 1`.

Si on rajoute une comme `print_int (factorielle 5)` et qu'on essaie de compiler, on obtient un message d'erreur :

```
File "test.ml", line 4, characters 16-27:
Unbound value factorielle
```

L'erreur se situe au niveau du nom `factorielle` qui se situe dans le corps de la fonction `factorielle`. Cette erreur est normale : on est en train de définir la fonction factorielle, donc elle n'est pas encore définie. Pour résoudre ce problème, il suffit de rajouter le mot clé `rec` juste après le `let`. Il permet d'indiquer que l'on définit une fonction récursive. Maintenant :

```
let rec factorielle n =
   if n = 1
      then 1
      else n * (factorielle (n-1))
   in

print_int (factorielle 5);
```

le code compile et s'exécute correctement, affichant le résultat 120.

Pour définir une fonction récursive, on place le mot clé `rec` entre le `let` et le nom de la fonction.

▶ On peut suivre l'ordre d'appel des fonctions en ajoutant un `print_int n` tout au début du corps de la fonction. Ainsi :

```
let rec factorielle n =
   print_int n;
   print_string " ";
```

```
   if n = 1
      then 1
      else n * (factorielle (n-1))
   in

print_int (factorielle 5);
```

affiche `5 4 3 2 1 120`.

6.3.3 Coffre fort en récursif

★ Reprenons le premier programme qu'on avait écrit avec une boucle `while`, celui qui simule le coffre fort.

```
while read_int() <> 36 do
   print_string "Code non valide !\n";
done;
print_string "Coffre ouvert !\n";
```

On va essayer d'écrire ce programme à l'aide d'une fonction récursive nommé `coffre_fort`.

- La fonction ne prend pas de paramètres.
- Si `read_int()` vaut 36, alors on affiche `"Coffre ouvert !"`.
- Sinon, on affiche `"Code non valide !"`, et on recommence, c'est-à-dire qu'on rappelle la fonction `coffre_fort`.

Voici le code qu'il faut compléter :

```
let rec coffre_fort () =
   (...insérer la solution...)
   in

coffre_fort();
```

_ _

```
let rec coffre_fort () =
   if read_int() = 36 then
      print_string "Coffre ouvert !\n"
   else
      begin
      print_string "Code non valide !\n";
      coffre_fort ()
      end;
   in
```

Remarque : si on veut, on peut nommer la valeur donnée par l'utilisateur avant de faire le test :

```
let rec coffre_fort () =
   let x = read_int() in
   if x = 36 then
   ...
```

★ Il s'agit d'une variante sur l'exercice du coffre fort. Pour augmenter la sécurité du coffre, on aimerait que le coffre se verrouille si l'utilisateur se trompe plus que `nb_restant` fois, où `nb_restant` est le nombre de tentatives autorisées.

Le code secret ainsi que le nombre total de tentatives autorisées sont données au début. Voici donc le code à compléter :

```
let code_secret = read_int() in
let nb_tentatives = read_int() in

(...définition de coffre_tres_fort avec le paramètre nb_restant...)

coffre_tres_fort nb_tentatives;
```

- La fonction `coffre_tres_fort` prend en paramètre un entier `k`.
- Si `nb_restant` vaut 0, cela veut dire qu'il ne reste plus de tentatives, on affiche `"Coffre bloqué !"`.
- Si l'utilisateur donne l'entier `secret`, on affiche `"Coffre ouvert !"`.
- Si l'utilisateur donne un autre entier, on affiche `"Code non valide !"`, et on recommence avec un nombre de tentatives autorisés diminué de 1. Pour cela, on appelle récursivement `coffre_tres_fort (nb_restant - 1)`.

De plus, toutes les chaînes affichées doivent être suivies d'un retour à la ligne pour que tout soit bien présenté.

– –

Pour distinguer les différents cas possibles, on utilise une structure avec des `else if`.

```
let code_secret = read_int() in
let nb_tentatives = read_int() in

let rec coffre_tres_fort nb_restant =
   if nb_restant = 0 then
      print_string "Coffre bloqué !\n"
   else if read_int() = code_secret then
      print_string "Coffre ouvert !\n"
   else
      begin
      print_string "Code non valide !\n";
      coffre_tres_fort (pred nb_restant)
      end
   in

coffre_tres_fort nb_tentatives;
```

★ Il s'agit d'écrire le même programme que pour Coffre fort avec limite en récursif, mais cette fois en utilisant une boucle `while` et pas de récursivité. Le code à compléter reste le même, et la fonction `coffre_tres_fort` ne devra plus être récursive.

– –

L'idée est d'utiliser une référence `r` pour compter le nombre d'essais restants. La boucle `while` s'arrête dès que le code secret est trouvé ou que `r` atteint 0. Le corps de la boucle consiste à afficher `"Code non valide"`, et à

faire décroître le contenu de `r`. Lorsqu'on quitte la boucle, ce n'est pas fini. Il faut encore déterminer pour quelle raison la boucle s'est terminé. Soit c'est parce qu'il n'y a plus d'essais, et dans ce cas le coffre se bloque, soit c'est parce que le code a été trouvé, et dans ce cas le coffre s'ouvre.

```
let code_secret = read_int() in
let nb_tentatives = read_int() in

let coffre_tres_fort nb_restant =
   let r = ref nb_restant in
   while (!r > 0) && (read_int() <> code_secret) do
      print_string "Code non valide !\n";
      decr r;
   done;
   if !r = 0
      then print_string "Coffre bloqué !\n"
      else print_string "Coffre ouvert !\n";
   in

coffre_tres_fort nb_tentatives;
```

6.3.4 Boucles en récursif

▶ Reprenons la fonction `affiche_entiers` qui prend en paramètre un entier `debut` et un entier `fin` et qui affiche tous les entiers compris entre ces deux valeurs incluses. On en a deux versions.

La première avec une boucle `for` :

```
let affiche_entiers debut fin =
   for i = debut to fin do
      print_int !i;
      print_string " ";
   done;
   in
```

Et la seconde avec une boucle `while` :

```
let affiche_entiers debut fin =
   let i = ref debut in
   while !i <= fin do
      print_int !i;
      print_string " ";
      incr i;
   done;
   in
```

On va en voir une troisième version sans aucune boucle, juste avec du récursif. Vocabulaire : les codes utilisant des boucles `for` ou `while` sont des codes dits `"itératifs"`.

On va transformer cette fonction en une fonction récursive, à partir du principe qui suit. Pour afficher tous les nombres entre `debut` et `fin`, si `debut <= fin`, on commence par afficher la valeur de `debut`, et ensuite on affiche tous les nombres entre `debut+1` et `fin`. Lorsque `debut > fin`, il n'y a rien à faire.

```
let rec affiche_entre debut fin =
   if debut <= fin then
```

```
      begin
      print_int debut;
      print_string " ";
      affiche_entre (succ debut) fin;
      end;
   in
```

Pour utiliser une fonction récursive, c'est exactement pareil que pour les autres. Un exemple :

```
affiche_entiers 23 54;
```

On n'a pas gagné beaucoup de lignes avec la version récursive, mais il n'y a plus de références. Par conséquent, le code est donc plus facile à comprendre (du moins une fois qu'on s'est habitué au principe du récursif).

6.3.5 Débordement de pile

► Pour déterminer la parité d'un entier `n`, on a vu que l'on pouvait utiliser l'expression `((n mod 2) = 0)` qui est `true` si et seulement si `n` est pair. On va ici écrire une fonction qui fait la même chose sans utiliser l'opérateur `mod`, en utilisant une fonction récursive.

Le principe est le suivant : si `n = 0`, on retourne vrai, et si `n` est strictement positif, on retourne l'opposé de la parité de `n-1`. Voici le code, que l'on teste sur l'entier 3.

```
let rec est_pair n =
   if n = 0
      then true
      else not (est_pair (pred n))
   in

print_string (string_of_bool (est_pair 3));
```

Pour trouver la parité de 3, on calcule récursivement celle de 2, puis celle de 1, et enfin celle de 0. Mathématiquement, comme 0 est pair, son suivant 1 est impair, et donc 2 est pair, et par conséquence 3 est impair. Du point de vue des robots, `est_pair 0` est `true`, donc `est_pair 1` est `not true`, c'est-à-dire `false`, et ainsi de suite...

► Essayez maintenant la fonction sur l'entier 100 millions :

```
let rec est_pair n =
   if n = 0
      then true
      else not (est_pair (pred n))
   in

print_string (string_of_bool (est_pair 100000000));
```

Vous obtiendrez le message d'erreur suivant :

```
Fatal error: exception Stack_overflow
```

Le code est tout à fait correct, et devrait théoriquement afficher `true`, mais là le programme plante. Cela est dû à la limitation de la mémoire : au bout d'un moment il n'y aura plus de place pour créer un nouveau robot, et le

programme génère une erreur. Plus précisément, la mémoire utilisée par un programme pour stocker les robots qu'il utilise s'appelle **la pile**. Lorsque la pile est pleine et qu'on veut rajouter encore un robot, elle déborde.

Un trop grand nombre d'appels récursifs imbriqués provoque un débordement de pile (stack overflow).

▶ Il est utile de savoir jusqu'où on peut aller. Par tâtonnements successifs, on s'aperçoit que `est_pair 65470` est la dernière valeur que l'on peut calculer. Attention, cette valeur `65470` dépend à la fois de la fonction récursive sur laquelle on travaille, mais aussi de la machine sur laquelle le programme est exécuté.

La limitation de la pile est une contrainte qu'il ne faut pas oublier, car après tout, une centaine de milliers est un petit nombre pour un ordinateur. Heureusement, il y a différentes solutions pour contourner ce problème comme on le verra plus tard.

6.3.6 Récursion infinie

▶ Pouvez-vous deviner ce qu'il va se passer si l'on fait :

```
let rec factorielle n =
   if n = 1
      then 1
      else n * (factorielle (n-1))
   in

print_int (factorielle (-3));
```

Testez aussi ce code pour voir ce qu'il se passe. Vous devriez obtenir encore un débordement de pile :

```
Fatal error: exception Stack_overflow
```

Suivons ce qui se passe. Un premier robot prend en paramètre l'entier `-3`. Comme `n` est différent de `1`, un robot assistant va être appelé. Cet assistant va prendre en paramètre l'entier `n - 1`, c'est-à-dire `-4`, et va faire appel à un troisième robot. Ce dernier prendra le paramètre `-5`, et appeler un quatrième robot, etc...

Comme `n` ne fait que décroître, il ne sera jamais égal à 1, et on va avoir besoin sans cesse de nouveau robots. Et fatalement, on fait tôt ou tard déborder la pile. La récursivité infinie est l'analogue de la boucle `while` infinie : la condition qui est censé faire qu'on s'arrête au bout d'un moment n'est jamais vérifiée.

Un stack overflow a souvent pour cause la récursivité infinie.

▶ Remarque : sans la contrainte de la mémoire, le programme s'arrêterait au bout d'un moment, et ce à cause de la limitation des entiers. Souvenez-vous que l'entier juste en dessous `-1073741824` est `1073741823`. Lorsqu'on part d'un négatif, et qu'on descend suffisamment longtemps, on finit par arriver sur 1.

6.4 Exercices

6.4.1 Syracuse en récursive

▶ On va réécrire dans cette partie les fonctions d'étude de la suite de Syracuse en récursif. Rappel de la fonction qui permet de passer d'une valeur à la suivante :

```
let syracuse p =
   if p mod 2 = 0
      then p / 2
      else 3 * p + 1
   in
```

★ Écrire une fonction récursive nommée `affiche_syracuse` prennant en paramètre un entier `n`. La fonction affiche d'abord la valeur de `n`, suivie d'un espace. Ensuite, si `n` est différent de 1, la fonction s'appelle récursivement avec le paramètre `(syracuse n)` de manière à afficher les termes suivants de la suite. Lorsque `n` vaut 1, on s'arrête.

_ _

Il n'y a qu'a traduire étape par étape :

```
let rec affiche_syracuse n =
   print_int n;
   print_string " ";
   if n <> 1
      then affiche_syracuse (syracuse n);
   in
```

Pour tester la fonction, on déclare les fonctions `Syracuse` et `affiche_Syracuse`, et on appelle la seconde avec le paramètre 7 par exemple :

```
let syracuse p =
   if p mod 2 = 0
      then p / 2
      else 3 * p + 1
   in

let rec affiche_syracuse n =
   print_int n;
   print_string " ";
   if n <> 1
      then affiche_syracuse (syracuse n);
   in

affiche_syracuse 7;
```

Affiche :

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

► Remarque : une autre approche consiste à effectuer d'abord le calcul de `Syracuse n`, que l'on nomme `etape`. On affiche alors la valeur de `etape`, puis si celle-ci est différente de 1, on appelle récursivement `affiche_Syracuse etape`.

```
let rec affiche_syracuse n =
   let etape = syracuse n in
   print_int etape;
   print_string " ";
   if etape <> 1
      then affiche_syracuse etape;
   in
```

Cette fonction n'est pas tout à fait équivalente à la précédente, puisqu'on n'affiche pas la première valeur. Ainsi pour `affiche_syracuse 7` :

```
22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Cette version n'est pas correcte car elle est fausse pour le paramètre `n=1`, vu qu'elle affiche `4 2 1`, alors qu'il n'y a rien besoin de faire. En effet, on effectue une étape avant de savoir s'il y a besoin d'en effectuer une !

★ Ecrivez la fonction récursive `etapes_syracuses` qui retourne le nombre d'étapes nécessaire pour arriver à 1 à partir du paramètre `n`. L'idée est simple. Si `n` vaut 1, il n'y a aucune étape à faire, et on retourne 0. Sinon, on renvoie le nombre d'étapes nécessaires pour arriver à 1 à partir de `etapes_Syracuses n`, augmenté de 1.

– –

```
let rec etapes_syracuses n =
   if n = 1
      then 0
      else succ (etapes_syracuses (syracuse n))
   in
```

► Cette fois la version récursive est beaucoup plus courte que l'itérative, que l'on redonne ici :

```
let etapes_syracuses depart =
   let nb_etapes = ref 0 in
   let n = ref depart in
   while !n <> 1 do
      n := syracuse !n;
      incr nb_etapes;
   done;
   !nb_etapes in
```

6.4.2 Descente en ascenseur

★ Un singe entre dans un ascenseur tout en haut d'une tour de 100 étages (il est donc à l'étage 99), et souhaite ne pas en sortir avant d'arriver au rez-de-chaussée (étage 0), d'où il pourra sortir de l'immeuble.

Il a de la chance, car l'ascenseur qu'il prend à la propriété de ne pouvoir que descendre : si on appuie sur un bouton qui désigne une destination plus haute que la position actuelle, il ne se passe rien. Et il ne se passe rien non plus si on appuie sur un bouton de l'ascenseur pendant que celui-ci se déplace.

Le singe n'étant pas très subtil, il appuie sur une touche au hasard parmi les 100 touches correspondant aux destinations possibles, jusqu'à ce qu'il arrive tout en bas. Votre objectif est d'écrire une fonction permettant de prévoir le nombre d'arrêts que va faire l'ascenseur avant d'arriver au rez-de-chaussée.

– –

C'est le même principe que pour compter le nombre d'étapes dans la suite de Syracuse. Si l'étage est 0, on renvoie 0, sinon on renvoie un de plus que le nombre d'arrêts qu'on fera après s'être déplacé une fois. La destination du premier déplacement que l'on effectue est `Random.int etage`, où

```
let rec nb_arrets etage =
   if etage = 0
      then 0
      else succ (nb_arrets (Random.int etage))
   in
```

On peut tester le programme avec :

```
Random.self_init();
print_int (nb_arrets 99);
```

Pour afficher les étages auxquels l'ascenseur s'arrête, il suffit d'insérer deux lignes d'affichage tout au début de la fonction :

```
let rec nb_arrets etage =
   print_int etage;
   print_string " ";
   if etage = 0
      then 0
      else succ (nb_arrets (Random.int etage))
   in
```

Tous les étages d'arrêts sont ainsi affichés, et le dernier entier précise le nombre d'arrêts.

Remarque : la mauvaise version :

```
let rec nb_arrets etage =
   let destination = Random.int etage in
   if destination = 0
      then 1
      else succ (nb_arrets destination)
   in
```

▶ Il existe une autre façon de retourner le nombre d'arrêts. Le principe est de donner en paramètre de la fonction le nombre d'arrêts qui ont déjà été effectués. Lorsqu'on effectue un appel récursif, on donne en paramètre le successeur du nombre d'arrêts déjà faits. Et lorsqu'on arrive à destination, on se contente alors de retourner cette valeur. Voici la fonction :

```
let rec nb_arrets etage etapes =
   let destination = Random.int etage in
   if destination = 0
      then etapes
      else nb_arrets destination (succ etapes)
   in
```

Pour l'appeler, il faut maintenant donner deux paramètres : l'étage duquel on part, et le nombre d'étapes déjà faites. On fera donc :

```
Random.self_init();
print_int (nb_arrets 99 0);
```

Vocabulaire : on dit que la variable `etape` est un **accumulateur**. On verra l'intérêt de cette technique plus tard, pour l'instant ça n'apporte rien, et même cela complique l'utilisation de la fonction puisqu'il y a un paramètre en plus.

★ Pouvez-vous écrire le même programme en itératif ? (c'est-à-dire avec une boucle `for` ou une boucle `while`).

_ _

Avec une boucle `while` qui s'effectue tant que l'étage est strictement supérieur à 0.

```
Random.self_init();
let etage = ref 99 in
let nb_arrets = ref 0 in
while !etage > 0 do
   incr nb_arrets;
   etage := Random.int !etage;
done;
print_int !nb_arrets;
```

6.4.3 Course aux dés en récursif

★ Cette fois, on va écrire en récursif le code de la course aux dés. Rappel : deux joueurs essaient tous les deux d'arriver à dépasser la case numéro 30.

Ecrivez une fonction récursive `course` prenant en paramètre la position des deux joueurs, et qui retourne le résultat de la course sous forme d'une chaîne : `"Joueur 1 gagne"`, ou `"Joueur 2 gagne"`, ou alors `"Joueurs ex-aequo"`. Testez alors le programme avec l'appel `course 0 0`.

– –

Appelons `joueur1` et `joueur2` les positions des deux joueurs. On compare ces valeurs à 30 pour déterminer ce qu'il faut retourner. Si aucun des joueurs n'est arrivé, on appelle récursivement la fonction en ajoutant la valeur d'un dé à la position de chacun des deux joueurs. On utilise une fonction `de()` pour simuler un lancer de dé.

```
let de () =
   (Random.int 6) + 1 in

let rec course joueur1 joueur2 =
   if (joueur1 >= 30) && (joueur2 >= 30) then
      "Joueurs ex-aequo"
   else if (joueur1 >= 30) then
      "Joueur 1 gagne"
   else if (joueur2 >= 30) then
      "Joueur 2 gagne"
   else
      course (joueur1 + de()) (joueur2 + de())
   in

Random.self_init();
print_string (course 0 0);
```

Remarque : on peut afficher les étapes du jeu en rajoutant au début de la fonction des instructions d'affichage :

```
let rec course joueur1 joueur2 =
   print_int joueur1;
   print_string " - ";
   print_int joueur2;
   print_newline();
   ...
```

► La méthode permettant de transformer un code avec une boucle `while` en un code avec une fonction récursive est toujours la même :

- Une référence est remplacée la valeur de retour de la fonction (c'était `status`).
- Les autres références deviennent des paramètres de la fonction (c'est `joueur1` et `joueur2`).
- La condition de la boucle devient la condition d'un test `if` pour savoir s'il faut effectuer un appel récursif.
- Le corps de la boucle `while` devient le corps de la fonction.
- Les valeurs initiales donnés aux compteurs deviennent les paramètres initiaux de l'appel à la fonction (ici on fait `course 0 0`).

6.4.4 "C'est plus, c'est moins" en récursif

★ Reprenez le jeu "c'est plus, c'est moins", et utiliser une fonction récursive afin de vous débarrasser des deux références et de la boucle `while`.

_ _

On commence par choisir une valeur cible au hasard, puis on utilise une fonction récursive `demande()` pour demander à l'utilisateur de deviner une valeur, et effectuer un affichage en fonction de cette valeur; La fonction retourne alors le nombre de coups qui ont été nécessaires.

Première méthode. Une première structure `if` permet de choisir quelle réponse afficher à l'utilisateur. Une seconde structure `if` permet de décider s'il faut continuer et renvoyer 1 plus le nombre de coups nécessaires à finir, ou alors s'il faut s'arrêter et renvoyer simplement 1.

```
Random.self_init();
let cible = Random.int 100 in

let rec demande () =

   print_string "Essayez un nombre : ";
   let essai = read_int() in

   if cible > essai then
      print_string "C'est plus\n"
   else if cible < essai then
      print_string "C'est moins\n"
   else
      print_string "C'est gagné !\n"
   ;

   if cible = essai
      then 1
      else succ (cible jeu)
   in

print_int (demande());
```

Seconde méthode. On utilise une grande structure `if`, dans laquelle chacun des blocs affiche le texte et donne la valeur à retourner.

```
let rec demande () =
   print_string "Essayez un nombre : ";
   let essai = read_int() in
   if cible > essai then
```

```
      begin
      print_string "C'est plus\n";
      succ (cible jeu)
      end
   else if cible < essai then
      begin
      print_string "C'est moins\n";
      succ (cible jeu)
      end
   else
      begin
      print_string "C'est gagné !\n";
      1
      end
   ;
   in
```

L'avantage de cette technique est de bien distinguer tous les cas. Mais son inconvénient, c'est qu'on est obligé d'écrire à deux endroits l'appel `succ (cible jeu)`.

Power is coming to you.

Chapitre 7

expression

7.1 Introduction

7.1.1 Objectif

► Ce chapitre précise le principe des expressions. Les expressions, c'est la vision que le compilateur a du code qu'on a écrit. Pour le compilateur, l'intégralité du code est construit avec des expressions. Comprendre comment le code est interprété par le compilateur a deux intérêts. Le premier est de pouvoir comprendre les messages d'erreurs qu'on peut obtenir. Le second est de pouvoir comprendre précisément de quelle manière sera interprété le code que l'on écrit. Détaillons ces deux points.

Des messages d'erreurs, on en a déjà vu un bon nombre. Lorsque le compilateur n'arrive pas à décortiquer le code en expression, il nous renvoie de sympathiques erreurs de syntaxes (`syntax error`), ou bien d'erreur de typage (`this expression has type ... but is here used with type ...`). Lorsqu'on utilise des noms qui ne sont associés à rien, on a le `unbound value`, facile à corriger. Il y a encore des erreurs d'utilisation de fonctions. Lorsqu'on utilise une valeur comme une fonction : `This expression is not a function, it cannot be applied`. Et lorsqu'on place plus d'arguments qu'il n'est censé en avoir : `This function is applied to too many arguments`.

L'étude des expressions nous apportera aussi une compréhension nécessaire de certaines difficultés de la programmation. La première arrive lorsqu'on essaie d'associer une valeur à un nom qui est déjà associé à une autre valeur. On appelle cela la surdéfinition de variable. Une deuxième difficulté est la construction de valeurs avec des structures `if` : on associe un nom à une valeur qui dépend d'un test. Troisième chose : connaître précisément le fonctionnement des expressions avec des `&&` et des `||` du calcul booléen. En dernier, on s'intéressera aux fonctions.

7.1.2 Principe

► Le principe de base est que chaque mot écrit dans le programme forme une expression, et que chaque expression est d'un et un seul type. On construit des expressions plus grande en groupant des petites expressions, puis des expressions encore plus grandes en regroupant ces grandes expressions, et ainsi de suite. A la fin, le code du programme tout entier forme une seule très grande expression. Avant de voir comment on peut construire de grandes expressions à partir de petites, il faut déjà commencer par voir les petites.

▶ Les **valeurs immédiates** sont les plus courtes expressions. On connaît les entiers, les flottants, les caractères, et les booléens :

- `5` est une expression de type `int`.
- `-4.67` est une expression de type `float`.
- `'c'` est une expression de type `char`.
- `true` est une expression de type `bool`.

▶ Les références, les tableaux et les chaînes peuvent être construits à partir de ces valeurs immédiates :

- `ref 5` est une expression de type `int ref`.
- `[| 3.4; 5.7; -2.34; 9.1 |]` est une expression de type `float array`.
- `"super"` est une expression de type `string`.
- `4 <> 5` est une expression de type `bool`.

Remarques : une chaîne est une succession de caractères placés entre des guillemets. Autre remarque : on verra plus tard que le mot `ref` est en fait le nom d'une fonction.

▶ Par définition, une expression qui réalise une action est de type `unit`.

Dans ce contexte, ce mot veut simplement dire `"action"`. Une telle expression n'a pas besoin d'être terminée par un point-virgule s'il n'y a rien après. En effet, le point-virgule ne sert qu'à séparer des expressions, comme on le précisera ensuite.

- `print_int 5` est une expression de type `unit`.
- `print_float 3.14` est une expression de type `unit`.
- `print_string "super"` est une expression de type `unit`.
- `print_newline()` est une expression de type `unit`.

Remarque : ces expressions sont l'application d'une fonction avec des paramètres. Il s'agit donc déjà d'expressions construites à partir de petits bouts.

Deux expressions permettent de dire de ne rien faire :

- `begin end` est une expression de type `unit`.
- `()` est une expression de type `unit`.

▶ Avec des fonctions prédéfinies, on peut demander à l'utilisateur de fournir des valeurs. Par exemple : `read_int` est une fonction de type `unit -> int`. Lorsqu'on applique la fonction avec le paramètre `()` représentant le type `unit`, ce qui donne `read_int()`, on obtient une expression de type `int`. Ainsi :

- `read_int()` est une expression de type `int`.
- `read_float()` est une expression de type `float`.
- `read_line()` est une expression de type `string`.

7.2 Fondements

7.2.1 Juxtaposition

▶ La règle que l'on va annoncer peut sembler déroutante au début, mais on se rendra vite compte qu'en fait on l'utilise déjà sans vraiment le savoir.

Une expression `"expr1; expr2"`, où `exrp1` est une expression de type unit, est de la valeur du type de l'expression `expr2`.

Tout de suite un exemple. On a vu juste avant que `print_string "test"` est une expression de type `unit`, et que 5 était une expression de type `int`. Donc d'après la règle, la juxtaposition des deux :

```
print_string "test"; 5
```

forme une expression de type `int`, qui vaut 5. Puisque cette expression représente l'entier 5, on va l'afficher avec un `print_int`, en prenant soin de l'entourer de parenthèses :

```
print_int (print_string "test"; 5)
```

Cette ligne forme donc une expression de type `unit` : c'est notre programme dans son ensemble.

▶ Regardons maintenant ce qu'il se passe lorsqu'on exécute ce programme.

Pour évaluer l'expression `expr1; expr2` où `exrp1` est une expression de type unit, on commence par faire l'action de `expr1`, puis on calcul la valeur de `expr2`.

Donc pour évaluer `print_string "test"; 5`, on commence par afficher `"test"`, et ensuite on donne la valeur 5. Ainsi le code :

```
print_int (print_string "test"; 5)
```

affiche `test5`.

▶ Remarque : on peut aussi utiliser une déclaration pour nommer l'expression `print_string "test"; 5`. Ce code est équivalent au précédent :

```
let x = print_string "test"; 5 in
print_int x
```

On précisera le fonctionnement du `let` dans la partie suivante.

▶ Revenons à la règle. Dans `expr1; expr2`, lorsque `expr1` et `expr2` sont tous les deux de type `unit`, puisque l'ensemble forme une expression du même type que `expr2`, il est donc de type `unit`.

Le code suivant est la juxtaposition de deux expressions de type `unit`, il fait quelque chose :

```
print_string "test"; print_newline()
```

7.2.2 Juxtaposition itérée

► Voyons maintenant ce qu'il se passe-t-il si l'on juxtapose 3 expressions de type `unit`. Par exemple :

```
print_string "test"; print_int 5; print_string "cinq"
```

C'est très simple, le compilateur regroupe les deux premières expressions :

```
( print_string "test"; print_int 5 ) ; print_string "cinq"
```

Il peut maintenant décortiquer correctement le code à l'aide de la règle énoncé en début de partie. D'abord :

```
print_string "test"; print_int 5
```

est la juxtaposition de deux expressions de type `unit`, donc une expression de type `unit`.

Ensuite, on va juxtaposer cette expression (qui est déjà elle-même une juxtaposition) avec l'expression `print_string "cinq"` qui est aussi de type `unit`. Au final,

```
print_string "test"; print_int 5 ; print_string "cinq"
```

est une expression de type `unit`. C'est un code valide, qui décrit un programme qui affiche `test5cinq`.

► Dernière chose à voir : une suite de plusieurs expressions de type `unit`, terminé par une expression d'un autre type que `unit`. Exemple :

```
print_string "test"; print_newline(); 5
```

est une expression de type `int`, qui vaut 5.

Pour aboutir à ce résultat, le compilateur à simplement lu :

```
( print_string "test"; print_newline() ) ; 5
```

D'abord

```
print_string "test"; print_newline()
```

est la juxtaposition de deux expressions de type `unit`, donc est de type `unit`.

Ensuite

```
( print_string "test"; print_newline() ) ; 5
```

est la juxtaposition d'une expression de type `unit` et d'une autre de type `int`, et est donc au final une expression de type `int`.

On peut affiche cette expression avec un `print_int` :

```
print_int ( print_string "test"; print_newline(); 5 )
```

Ce code affiche `"test"`, puis un retour à la ligne, puis 5 :

```
test
5
```

▶ On peut généraliser ces deux constructions en disant que :

```
expr1; expr2; ... exprN-1; exprN
```

où tous les termes de `expr1` à `exprN-1` sont forcément de type `unit` constitue une expression dont la valeur et le type est celui de `exprN`.

Si une des expressions entre `expr1` et `exprN-1` n'est pas de type `unit`, on obtient une erreur. Un exemple :

```
print_string "test"; 5 ; print_string "cinq"
```

```
File "test.ml", line 1, characters 21-22:
Warning: this expression should have type unit.
```

Le problème est localisé sur le `5` qui devrait être de type `unit`, mais qui ne l'est pas, puisque 5 est un entier.

Il ne s'agit pas d'une erreur, mais seulement d'un avertissement (`warning`). Le code fonctionne donc, la valeur 5 est simplement ignorée, et le programme affiche : `testcinq`.

7.2.3 Bloc d'expression

▶ On peut forcer un certain groupement d'expressions avec des parenthèses ouvrante puis fermante, ou bien avec les bornes `begin` et `end`.

Ainsi `begin expr1 end` comme `(expr1)` sont simplement équivalents à `expr1`.

On a déjà vu un exemple :

```
print_int (print_string "test"; 5)
```

Voici le même avec `begin` et `end` :

```
print_int begin print_string "test"; 5 end
```

On reviendra sur les blocs au moment des problèmes de surdéfinition, et aussi pour les structures `if`.

7.2.4 Déclaration

▶ Énonçons une seconde règle fondamentale, qui pourra elle-aussi sembler un petit peu déroutante au début.

L'expression `let name = expr1 in expr2` vaut `expr2` dans laquelle `name` est un nom qui peut être utilisé dans `expr2`, et qui représente la valeur de `expr1` qu'on calcul une fois pour toutes.

Remarque : le nom `name` est associée à une valeur uniquement dans `expr2`. Mais ce nom n'est associé à rien dans `expr1` ou ailleurs (à moins qu'il y ait ailleurs une autre définition associant `name` à une valeur).

▶ Reprenons un de nos tout premiers programmes, et décortiquons-le comme le fait le compilateur :

```
let x = 238 in
print_int x;
print_string " ";
print_int (x * x)
```

Explications. Le nom associé à une valeur est `x`. La valeur associé à ce nom est `238`. L'expression dans laquelle le nom `x` est associé à la valeur `238` est l'expression :

```
print_int x; print_string " "; print_int (x * x)
```

C'est une juxtaposition de trois expressions de type `unit`, donc une expression de type `unit`. L'ensemble du programme est donc aussi une expression de type `unit`.

Remarque : le programme peut être terminé par un point-virgule si on veut, mais il n'y en a pas besoin puisqu'il n'y a rien après.

▶ Décortiquons maintenant un autre programme, avec deux `let` :

```
let x = read_int() in
let y = read_int() in
print_int (x + y)
```

Il y a deux déclarations. Dans la première, on associe le nom `x` à une première valeur donnée par l'utilisateur, et ce à l'intérieur de l'expression :

```
let y = read_int() in
print_int (x + y)
```

Dans cette seconde déclaration, on associe le nom `y` à une seconde valeur donnée par l'utilisateur, et ce à l'intérieur de l'expression :

```
print_int (x + y)
```

Cette dernière expression est de type `unit`. Donc

```
let y = read_int() in
print_int (x + y)
```

est aussi de type `unit`. Et par conséquent :

```
let x = read_int() in
let y = read_int() in
print_int (x + y)
```

est une expression de type `unit`, qui forme la totalité du programme.

Regardons alors ce qui se passe lors de l'évaluation. D'abord on donne une valeur à `x`, par exemple 5. Ainsi le code est équivalent à :

```
let y = read_int() in
print_int (5 + y)
```

Ensuite on donne une valeur à `y`, par exemple 8. Il reste :

```
print_int (5 + 8)
```

Ce code affiche 13, qui est le résultat attendu du programme.

▶ Dans certains cas, il peut y avoir une ambiguïté sur la manière de lire les expressions. Il y a ainsi deux façons d lire :

```
let x = read_int() in
print_int x;
let y = read_int() in
print_int y;
```

La première méthode, comme déclaration dans un bloc :

```
let x = read_int() in
begin print_int x; let y = read_int() in print_int y end
```

La seconde méthode, comme juxtaposition de deux blocs contenant chacun une déclaration :

```
begin let x = read_int() in print_int x end;
begin let y = read_int() in print_int y end
```

Dans une telle situation, le compilateur a une méthode pour choisir un des deux découpages possibles. Mais comme on n'a pas besoin de connaître cette méthode en détails, on n'en dira pas plus. De toutes façons, le résultat dans les deux cas est équivalent.

7.3 Structures

7.3.1 Décision

▶ L'expression `if expr1 then expr2 else expr3` où `expr1` est une expression de type `bool`, et où les expressions `expr2` et `expr3` sont forcément de même type, constitue une structure décisionnelle. Lorsque la condition `expr1` vaut `true`, la structure est équivalente à `expr2`, et dans le cas contraire lorsque `expr1` vaut `false`, la structure est équivalente à `expr3`.

▶ D'abord dans le cas où `expr2` et `expr3` sont de type `unit` :

```
let x = read_int() in
if x >= 0
   then print_string "positif"
   else print_string "négatif"
```

Ici, la condition est `x >= 0`. Les expressions `print_string "positif"` et `print_string "négatif"` sont toutes les deux de type `unit`.

▶ Lorsque les expressions `expr2` ou `expr3` sont construites comme juxtapositions de plusieurs expressions, il faut délimiter le bloc. Cela se fait de préférence avec `begin` et `end`, bien qu'on puisse aussi utiliser des parenthèses en théorie.

Voici l'exemple précédent dans lequel on a rajouté l'affichage de la valeur absolue du nombre lorsque celui-ci est négatif :

```
let x = read_int() in
if x >= 0 then
   print_string "positif"
else
   begin
   print_string "négatif, de valeur absolue : ";
   print_int (abs x)
   end
```

► Maintenant dans le cas où `expr2` et `expr3` ne sont pas de type `unit` : ces deux expressions doivent être de même type, et ce type est alors celui de l'ensemble de la structure `if`.

Par exemple, lorsque `x` est défini, l'expression :

```
if x >= 0 then "positif" else "négatif"
```

est de type `string`, puisque `"positif"` et `"négatif"` sont tous les deux de type `string`.

Ainsi :

```
let x = read_int() in
print_string (if x >= 0 then "positif" else "négatif");
```

affiche le signe de l'entier `x` donné par l'utilisateur.

On peut aussi associé un nom à l'expression formée par la structure `if` :

```
let x = read_int() in
let signe = if x >= 0 then "positif" else "négatif" in
print_string signe
```

Dans ce cas, il sera préférable de présenter cela sur plusieurs lignes, comme on présente les fonctions :

```
let x = read_int() in
let signe =
   if x >= 0
      then "positif"
      else "négatif"
   in
print_string signe
```

► Voyons un cas où `expr2` et `expr3` où sont des juxtapositions d'expressions, d'un autre type que `unit`. En fait on l'a déjà utilisé, pour le jeu "C'est plus, c'est moins" en récursif :

```
if cible > essai then
   begin
   print_string "C'est plus\n";
   succ (cible jeu)
   end
else if cible < essai then
   begin
   print_string "C'est moins\n";
```

```
    succ (cible jeu)
    end
else
    begin
    print_string "C'est gagné !\n";
    1
    end
;
```

▶ Terminons avec le bloc du `else` implicite.

L'expression `if expr1 then expr2` est équivalente à `if expr1 then expr2 else ()`. Par conséquent, `expr1` doit être de type `bool` et `expr2` doit être de type `unit`.

Par exemple :

```
let x = read_int() in
if x > 0
    then print_string "positif"
```

est équivalent à

```
let x = read_int() in
if x > 0
    then print_string "positif"
    else ()
```

Si `expr2` n'est pas de type `unit`, cela provoque une erreur de type. Illustration :

```
let x = read_int() in
print_string (if x > 0 then "positif")
```

```
File "test.ml", line 2, characters 28-37:
This expression has type string but is here used with type unit
```

Erreur sur la chaîne `"positif"` qui n'est pas de type `unit`.

7.3.2 Calcul booléen

▶ Le ET du calcul booléen, opérateur `&&`, est en fait équivalent à une structure avec des `if`. Cherchons l'équivalent de `expr1 && expr2`. Si `expr1` est vrai, alors si `expr2` est aussi vrai, le résultat est vrai. Mais si `expr1` ou `expr2` est faux, le résultat est faux. Ceci s'écrit :

```
if expr1 then
    begin
    if expr2
       then true
       else false
    end
else
    false
```

Mais l'expression `if expr2 then true else false` est équivalente à la valeur de `expr2` directement. Donc en simplifiant le code et la présentation, il reste :

```
if expr1
   then expr2
   else false
```

Conclusion :

L'expression `expr1 && expr2` où les deux `expr1` et `expr2` sont de type `bool` est équivalente à `if expr1 then expr2 else false`.

▶ Il y a une observation importante à faire : si `expr1` est évalué à faux, alors `expr2` ne sera même pas évalué, puisqu'on est déjà sûr que le résultat de `expr1 && expr2` est faux.

Pour vérifier cela, on va utiliser des expressions qui affichent quelque chose lorsqu'elles sont évaluées. Pour commencer, on va ainsi prendre pour `expr1` l'expression `print_string "expr1 "; true` et pour `expr2` l'expression `print_string "expr2 "; true`, et voir ce qu'il se passe en affichant le résultat de `expr1 && expr2` :

```
let b = (print_string "expr1 "; true) && (print_string "expr2 "; true) in
print_string (string_of_bool b);
```

Ce code affiche `expr1 expr2 vrai`. On en déduit que `expr1` a été évalué, puis `expr2`, et que `expr1 && expr2` était vrai.

Regardons maintenant ce qu'il se passe si on fait que `expr2` soit faux :

```
let b = (print_string "expr1 "; false) && (print_string "expr2 "; true) in
print_string (string_of_bool b);
```

Ce code affiche cette fois `expr1 faux`. On en déduit que `expr1` a été évalué, mais pas `expr2`, et que le résultat de `expr1 && expr2` était faux.

▶ On va calquer le raisonnement effectué pour le ET pour traiter le OU. L'expression `expr1 || expr2` vaut vrai lorsque l'une ou l'autre des deux expressions est vraie. Traduisons cela avec des `if`. Si `expr1` est vrai, on est sûr que le résultat est vrai. Sinon il faut regarder si `expr2` est vrai ou faux.

On a donc :

```
if expr1 then
   true
else
   begin
   if expr2
      then true
      else false
   end
```

qui se simplifie tout de suite en :

```
if expr1
   then true
   else expr2
```

Conclusion :

L'expression `expr1 || expr2` où les deux `expr1` et `expr2` sont de type `bool` est équivalente à `if expr1 then true else expr2`.

Vous pouvez vérifier que lorsque la condition `expr1` est vrai, la condition `expr2` n'est même pas évaluée.

7.3.3 Boucles

▶ La structure :

```
for compteur = expr1 to expr2 do
   expr3
done
```

où

- `compteur` est le nom du compteur,
- `expr1` et `expr2` sont des entiers, valeurs initiale et finale.
- `expr3` est une expression de type unit où l'on peut utiliser le nom du compteur, le corps de la boucle,

forme une expression de type `unit`, qui se comporte comme la répétition de `expr3` pour toutes les valeurs de `compteur` comprises entre `expr1` et `expr2` inclus. S'il n'existe pas de telle valeur pour le compteur, alors la boucle ne fait rien et est équivalente à `()`.

Remarque : les valeurs initiales et finales du compteur, `expr1` et `expr2`, ne sont évaluées qu'une seule fois au début. La preuve :

```
for i = (print_string "expr1 "; 1) to (print_string "expr2 "; 5) do
   print_int i;
done;
```

affiche `expr1 expr2 12345`.

▶ La structure :

```
while expr1 do
   expr2
done
```

où `expr1` (la condition de la boucle) est de type `bool` et `expr2` (le corps de la boucle) de type `unit` est une expression de type `unit`, qui se comporte comme la répétition de `expr2` tant que l'expression `expr1` est évaluée à vrai.

Par exemple :

```
let i = ref 1 in
while (print_string "expr1 "; !i <= 5) do
   print_int !i;
   incr i;
done;
```

affiche `"expr1 1 expr1 2 expr1 3 expr1 4 expr1 5 expr1"`. Ainsi l'expression `expr1` qui détermine l'exécution de la boucle est exécuté tout au début, mais aussi tout à la fin : la fois où elle est évaluée à faux.

7.3.4 Fonctions

▶ La définition d'une fonction :

```
let arg_de_la_fonction arg_1 arg_2 ... arg_N =
   expr1 in
```

où `arg_de_la_fonction` et tous les `arg_i` sont des noms, forme une expression de type :

```
type(arg_1) -> type(arg_2) -> ... -> type(arg_N) -> type(expr_1)
```

les types des arguments étant déduits de l'utilisation qui en est faîte dans `expr1`.

▶ L'expression :

```
nom_de_fonction expr_1 expr_2 ... expr_N
```

lorsque `nom_de_fonction` est associé à une fonction de type

```
type_arg_1 -> type_arg_2 -> ... -> type_arg_N -> type_retour
```

est une expression de type `type_retour`, à condition que chaque `expr_i` soit de type `type_arg_i`.

7.4 Ordre d'évaluation

7.4.1 Arguments

▶ Étudions ce qui se passe lorsqu'on appelle une fonction avec des arguments. Prenons une fonction a au moins deux arguments, qui réalise une tâche quelconque. Par exemple l'affichage de la somme des deux paramètres :

```
let f x y =
   print_int (x + y);
   in

f (3+2) (6-5);
```

Question très intéressante : calcul-t-on d'abord `(3+2)` ou `(6-5)` ? Ajoutons des opérations d'affichage pour le savoir :

```
let f x y =
   print_int (x + y);
   in

f (print_string "calcul a\n"; 3+2) (print_string "calcul b\n"; 6-5);
```

```
calcul b
calcul a
6
```

Conclusion : contrairement à l'ordre naturel, on calcul d'abord le second argument. Mais aucune règle ne garanti que ça restera comme ça.

Les arguments d'une fonction sont tous évalués avant l'exécution de la fonction. L'ordre d'évaluation des arguments n'est pas spécifié.

▶ Comment faire pour forcer un ordre particulier de calcul ? C'est très simple, on utiliser des `let` pour nommer les arguments. Dans notre exemple, on ferai :

```
let a = 3+2 in
let b = 6-5 in
f a b ;
```

La preuve :

```
let a = (print_string "calcul a\n"; 3+2) in
let b = (print_string "calcul b\n"; 6-5) in
f a b;
```

```
calcul a
calcul b
6
```

Pour préciser un ordre de calcul des arguments lors de l'application d'une fonction, on les définit auparavant avec un `let` dans l'ordre souhaité.

7.4.2 Fonction récursives

▶ On va maintenant reprendre la fonction récursive `factorielle` et rajouter des fonctions d'affichage afin de bien suivre la création et la destruction de tous les robots utilisés dans le processus. Ainsi on affiche `i` suivi de `"début"` lorsqu'un robot chargé de calculé `factorielle i` est mis en route, et `i` suivi de `"fin"` lorsque ce robot s'apprête à déposer son résultat et à s'auto-détruire. Comme déposer la valeur de retour est la dernière chose que fait un robot dans sa vie, on est obliger de nommer d'abord le résultat, pour avoir le temps d'afficher `"fin"` avant de retourner ce résultat.

```
let rec factorielle n =
   print_int n;
   print_string " debut\n";
   let resultat =
      if n = 1
         then 1
         else n * (factorielle (n-1))
      in
   print_int n;
   print_string " fin\n";
   resultat in

print_int (factorielle 3);
```

```
3 debut
2 debut
1 debut
1 fin
2 fin
3 fin
6
```

Un peu théorique, mais néanmoins indispensable.

Chapitre 8

Annexe : clavier français

8.1 Le clavier

► Les lettres minuscules peuvent être entrées directement à l'aide de la touche associée. Pour obtenir une lettre majuscule, il suffit que la touche majuscule soit enfoncé pendant que l'on appuie sur la touche de la lettre. Pour les caractères spéciaux, c'est un peu plus compliqué, car chaque touche est associé à deux ou trois caractères. Ces caractères sont dessinés sur la touche.

- Le caractère en bas à gauche est accessible directement en tapant sur la touche.
- Le caractère en haut à gauche est accessible en enfonçant la touche majuscule pendant que l'on appuie sur la touche.
- Le caractère en haut à droite est accessible en enfonçant la touche AltGr pendant que l'on appuie sur la touche.

► Il est recommandé d'étudier le clavier pendant une ou deux minutes avant de commencer à programmer, plutôt que de se reporter à l'annexe clavier trop souvent.

8.2 Opérateurs

=	égale	deux touches à droite du 0
+	plus	deux touches à droite du 0
-	moins	touche du 6
*	fois	au milieu à droite
/	divisé	en bas à droite

<	inférieur	en bas à gauche
>	supérieur	en bas à gauche

8.3 Séparateurs

```
_   undersore                touche du 8
\   backslash                touche du 8
```

```
;   point-virgule            en bas à droite
,   virgule                  en bas à droite
.   point                    en bas à droite
:   deux-points              en bas à droite
!   point d'exclamation      en bas à droite
?   point d'interrogation    en bas à droite
```

8.4 Délimiteurs

```
(   parenthèse ouvrante      touche du 5
)   parenthèse fermante      à droite du 0
[   crochet ouvrant          touche du 5
]   crochet fermant          à droite du 0
{   accolade ouvrante        touche du 4
}   accolade fermante        deux touches à droite du 0
```

```
"   guillemet double         touche du 3
'   guillemet simple         touche du 4
`   accent (sert à rien)     touche du 7
```

8.5 Avancé

```
&   et commercial            touche du 1
|   barre verticale          touche du 6
^   accent circonflexe       touche du 9
@   arobase                  touche du 0
#   dièse                    touche du 3
```

Chapitre 9

Annexe : résumés

9.1 Chapitre 0

▶ Conseils de lecture :

- lisez tous les paragraphes, et faites tous les exercices, sans exception;
- si vous bloquez sur un exercice, cherchez au moins 10 minutes avant de regarder la correction, mais ne passez jamais plus de 20 minutes sur un petit exercice (regardez votre montre);
- lorsque vous avez fini un exercice, retenez bien la solution, elle servira presque sûrement dans la suite;
- essayez de présenter vos programmes de la même manière que dans le cours;
- évitez de faire du copier-coller de code, car c'est en écrivant du code que l'on apprend.

9.2 Chapitre 1

▶ Pour effectuer plusieurs instructions, il suffit de les écrire à la suite les unes des autres.

▶ On écrira une seule instruction par ligne.

▶ La règle d'exécution d'un `"let nom = ... in"` est la suivante :

1. calcule la valeur de l'expression entre le signe `=` et le `in`,
2. remplace le nom par cette valeur dans les instructions concernées,
3. passe alors à l'exécution de la suite.

▶ Un nom de valeur doit être formé de lettres de bases (de a à z ou de A à Z), de chiffres, et d'underscores (le symbole _), et doit en outre commencer par une lettre minuscule. Il ne faut pas mettre d'espaces, ni de lettres accentuées ou tout autre caractère spécial.

▶ Un `read_int()` bloque l'exécution du programme jusqu'à ce que l'utilisateur ait tapé un nombre entier, et ait appuyé sur la touche entrée du clavier. A ce moment le mot `read_int()` est remplacé par la valeur donnée.

▶ Pour demander des données à l'utilisateur, on en posera toujours la valeur avec un `let`.

▶ `read_line()` permet de demander à l'utilisateur une ligne de texte.

▶ Fonctionnement de la structure `if, then, else` :

- On teste la condition qui se trouve après le `if`.
- Si ce test est vérifié, on exécute juste le bloc du `then`.
- Dans le cas contraire, on exécute juste le bloc du `else`.
- Dans les deux cas, on continue ensuite l'exécution après le point-virgule finale.

▶ Une association effectuée dans un des blocs d'une structure en `if` a une portée limitée à ce bloc.

▶

```
begin
instruction;
end
```

se simplifie simplement en `"instruction"`.

▶ Dans une structure `if`, lorsque les blocs du `then` et du `else` sont réduits à une instruction chacun, on adopte la présentation suivante :

```
if (...condition...)
   then (...instruction du then...)
   else (...instruction du else...);
```

▶ Un bloc vide `begin end` peut s'écrire aussi comme un couple de parenthèses : `()`.

▶ Dans une structure `if`, on peut se passer du `"else ()"`.

▶ Lorsque le bloc du `else` contient lui-même une structure `if`, on peut enlever les délimiteurs du bloc `begin` et `end`. Dans ce cas, il faut aussi enlever le point-virgule terminant le bloc imbriqué.

▶ Voici une structure avec des `else if` :

```
if (...1ere condition...) then
   (...bloc du 1er then...)
else if (...2ème condition...) then
   (...bloc du 2eme then...)
else if (...3ème condition...) then
   (...bloc du 3eme then...)
...
...
else
```

```
    ( ...bloc du else...)
;
```

► Fonctionnement de la boucle `for`, utilisée pour répéter un bloc :

1. La première ligne permet de préciser le nombre de fois qu'il faut répéter la boucle.
2. Les bornes `do` et `done` délimitent le bloc qu'on va répéter.
3. Le corps de la boucle `for` sera copié autant de fois qu'indiqué à la première ligne.
4. L'ensemble de la structure, formant une grosse instruction, doit être terminé par un point-virgule.

► Les déclarations réalisées à l'intérieur du corps de la boucle ont une portée limitée à ce corps.

► Fonctionnement de la boucle `for` utilisé pour itérer un bloc avec un compteur :

1. Évaluer la valeur initiale et la valeur finale du compteur, ce sont les bornes.
2. Prendre dans l'ordre croissant tous les entiers compris entre ces bornes, incluses.
3. Pour chaque entier, copier le corps du for, et y déclarer le compteur égal à cet entier.

► Lorsque la valeur initiale du compteur est égale à la valeur finale, la boucle n'est exécutée qu'une seule fois pour cette valeur. Lorsque la valeur initiale est supérieure strictement à la valeur finale, la boucle n'est exécutée aucune fois.

► Remplacer `to` par `downto` dans une boucle `for` pour avoir des valeurs du compteur décroissantes.

9.3 Chapitre 2

► Il est impossible de mélanger des int et des float sans soins particuliers

► Lorsqu'on effectue une division par zéro en calculant sur des réels, un code spécial est utilisé pour désigner le résultat. Sous Windows :

- `1.#INF` et `-1.#INF` respectivement pour plus et moins l'infini.
- `1.#IND` et `-1.#IND` pour des valeurs indéterminées.

Sous linux :

- `inf.` et `-inf.` respectivement pour plus et moins l'infini.
- `nan.` et `-nan.` pour des valeurs indéterminées.

► Pour calculer la valeur absolue d'un réel `x`, on fait `abs_float x`.

► Pour prendre la racine carrée d'un nombre réel `x`, on fait `sqrt x`.

► La fonction `max` donne le plus grand de deux nombres réels.

► L'opérateur `/` représente la division entière.

► L'opérateur `mod` calcule le reste de la division entière.

► La fonction `float_of_int` convertit un entier en le réel correspondant.

► Un `int` est une valeur comprise entre `-1073741824` et `1073741823`.

► Lorsqu'on travaille avec des `int`, il faut faire attention à ne pas dépasser les limites, car aucun message d'erreur ne signale les dépassements.

► Pour créer une boite, il faut donner un contenu initial. A tout instant, la boite contiendra un et un seul objet du même type que ce contenu initial.

► Résumé des manipulations de références :

- `let nom = ref contenu in` pour construire une nouvelle référence.
- `!nom` pour accéder au contenu de la référence.
- `nom := nouveau_contenu` pour affecter un nouveau contenu à la référence.

9.4 Chapitre 3

► Pour nommer une fonction, on choisit toujours un nom décrivant précisément l'action réalisée par le corps de cette fonction.

► Pour définir une fonction, on écrit :

```
let (...nom de la fonction...) (...noms des arguments...) =
   (...instructions du corps de la fonction...)
   in
```

► Pour appeler une fonction qui ne retourne rien, on écrit :

```
(...nom de la fonction...) (...valeurs des arguments...) ;
```

► Lorsqu'il y a plusieurs paramètres à une fonction, on les place simplement séparés par des espaces.

► Lorsqu'il n'y a aucun paramètre à une fonction, on place un couple de parenthèses.

▶ Le schéma d'une fonction qui retourne une valeur est le suivant :

```
let (...nom de la fonction...) (...noms des paramètres...) =
   (...actions réalisées par la fonction...)
   (...valeur de retour...) in
```

▶ La valeur de retour d'une fonction peut être le résultat d'une structure `if` :

```
let (...nom de la fonction...) (...noms des paramètre...) =
   (...actions de la fonction...)
   if (...condition...)
      then (...valeur de retour 1...)
      else (...valeur de retour 2...)
   in
```

Cette structure fonctionne aussi avec les `else if`.

▶ Le schéma général d'une fonction est le suivant :

```
let (...nom de la fonction...) (...noms des paramètres...) =
   (...actions réalisées par la fonction...)
   (...valeur de retour...) in
```

- si la fonction n'a pas d'argument, on place juste un couple de parenthèses;
- il peut n'y avoir aucune action réalisées par la fonction;
- si la fonction ne retourne rien, il n'y a pas de valeur de retour avant le `in`.
- la valeur de retour peut être le résultat d'un test.

9.5 Chapitre 4

▶ Résumé des manipulations de tableaux :

- Création d'un tableau, première version : `[| elem1; elem2; ... elemN |]`
- Création d'un tableau, seconde version : `Array.make taille motif`
- Accès à l'élément d'indice `i` : `tab.(i)`
- Modification de l'élément d'indice `i` : `tab.(i) <- nouvel_valeur`
- Longueur d'un tableau : `Array.length tab`
- Accès à un indice invalide : `exception Invalid_argument("Array.get")`
- Modification à un indice invalide : `exception Invalid_argument("Array.set")`

▶ La fonction Array.copy prend en paramètre un tableau et retourne un autre tableau, copie conforme du premier.

▶ Un caractère est une valeur de type `char` qu'on construit en plaçant le caractère entre des guillemets simples.

▶ Résumé des manipulations de chaînes :

- Création d'une chaîne, première version : `"mon texte"`
- Création d'une chaîne, seconde version : `String.make taille un_caractère`
- Longueur d'une chaîne : `String.length str`
- Accès à un caractère : `str.[i]`
- Modification d'un caractère : `str.[i] <- nouveau_caractère`
- Copie d'une chaîne : `String.copy str`.
- Concaténation de deux chaînes : `str1 ^ str2`.

▶ On peut convertir un nombre entier en la chaîne qui le représente à l'aide de la fonction `string_of_int`.

▶ `string_of_float` permet de convertir un nombre flottant en la chaîne qui le représente.

9.6 Chapitre 5

▶ Une valeur booléenne (`bool`) vaut `true` ou `false`, et peut se construire avec des tests comparatifs.

▶ La condition d'un `if` est toujours une expression de type `bool`.

▶ Pour afficher des booléens, on définit généralement la fonction `print_bool` au préalable :

```
let print_bool b =
   print_string (string_of_bool b);
   in
```

▶ Résumé des opérateurs de calcul sur les expressions booléennes :

- `(expr1 && expr2)` est vraie uniquement lorsque les deux expressions sont vraies.
- `(expr1 || expr2)` est vraie dès qu'une des deux expressions est vraie.
- `(not expr1)` est vraie lorsque l'expression est fausse, et fausse lorsque l'expression est vraie.

▶ `Random.int k` donne aléatoirement un entier compris entre 0 et `k-1`.

▶ L'instruction `Random.self_init()` doit être exécutée une fois pour toutes avant de pouvoir appeler le générateur de nombres aléatoires.

▶ La fonction `Random.float x` retourne un nombre aléatoire de type `float` compris entre zéro (inclus) et `x` (exclus).

▶ `Random.bool()` a une chance sur deux de renvoyer `true`, et une chance sur deux de renvoyer `false`

9.7 Chapitre 6

▶ Voici le schéma de la boucle `while` :

```
while (...condition...) do
   (...corps de la boucle while...)
done;
```

▶ Une fonction récursive est une fonction qui s'appelle elle-même.

▶ Pour définir une fonction récursive, on place le mot clé `rec` entre le `let` et le nom de la fonction.

▶ Un trop grand nombre d'appels récursifs imbriqués provoque un débordement de pile (stack overflow).

▶ Un stack overflow a souvent pour cause la récursivité infinie.

9.8 Chapitre 7

▶ Par définition, une expression qui réalise une action est de type `unit`.

▶ Une expression `"expr1; expr2"`, où `exrp1` est une expression de type unit, est de la valeur du type de l'expression `expr2`.

▶ Pour évaluer l'expression `expr1; expr2` où `exrp1` est une expression de type unit, on commence par faire l'action de `expr1`, puis on calcul la valeur de `expr2`.

▶ On peut forcer un certain groupement d'expressions avec des parenthèses ouvrante puis fermante, ou bien avec les bornes `begin` et `end`.

▶ L'expression `let name = expr1 in expr2` vaut `expr2` dans laquelle `name` est un nom qui peut être utilisé dans `expr2`, et qui représente la valeur de `expr1` qu'on calcul une fois pour toutes.

▶ L'expression `if expr1 then expr2 else expr3` où `expr1` est une expression de type `bool`, et où les expressions `expr2` et `expr3` sont forcément de même type, constitue une structure décisionnelle. Lorsque la condition `expr1` vaut `true`, la structure est équivalente à `expr2`, et dans le cas contraire lorsque `expr1` vaut `false`, la structure est équivalente à `expr3`.

▶ L'expression `if expr1 then expr2` est équivalente à `if expr1 then expr2 else ()`. Par conséquent, `expr1` doit être de type `bool` et `expr2` doit être de type `unit`.

▶ L'expression `expr1 && expr2` où les deux `expr1` et `expr2` sont de type `bool` est équivalente à `if expr1 then expr2 else false`.

▶ L'expression `expr1 || expr2` où les deux `expr1` et `expr2` sont de type `bool` est équivalente à `if expr1 then true else expr2`.

▶ La structure :

```
for compteur = expr1 to expr2 do
   expr3
done
```

où

- `compteur` est le nom du compteur,
- `expr1` et `expr2` sont des entiers, valeurs initiale et finale.
- `expr3` est une expression de type unit où l'on peut utiliser le nom du compteur, le corps de la boucle,

forme une expression de type `unit`, qui se comporte comme la répétition de `expr3` pour toutes les valeurs de `compteur` comprises entre `expr1` et `expr2` inclus. S'il n'existe pas de telle valeur pour le compteur, alors la boucle ne fait rien et est équivalente à `()`.

▶ La structure :

```
while expr1 do
   expr2
done
```

où `expr1` (la condition de la boucle) est de type `bool` et `expr2` (le corps de la boucle) de type `unit` est une expression de type `unit`, qui se comporte comme la répétition de `expr2` tant que l'expression `expr1` est évaluée à vrai.

▶ La définition d'une fonction :

```
let arg_de_la_fonction arg_1 arg_2 ... arg_N =
   expr1 in
```

où `arg_de_la_fonction` et tous les `arg_i` sont des noms, forme une expression de type :

```
type(arg_1) -> type(arg_2) -> ... -> type(arg_N) -> type(expr_1)
```

les types des arguments étant déduits de l'utilisation qui en est faîte dans `expr1`.

▶ L'expression :

```
nom_de_fonction expr_1 expr_2 ... expr_N
```

lorsque `nom_de_fonction` est associé à une fonction de type

```
type_arg_1 -> type_arg_2 -> ... -> type_arg_N -> type_retour
```

est une expression de type `type_retour`, à condition que chaque `expr_i` soit de type `type_arg_i`.

▶ Les arguments d'une fonction sont tous évalués avant l'exécution de la fonction. L'ordre d'évaluation des arguments n'est pas spécifié

▶ Pour préciser un ordre de calcul des arguments lors de l'application d'une fonction, on les définit auparavant avec un `let` dans l'ordre souhaité

9.9 Chapitre 8

► Il est recommandé d'étudier le clavier pendant une ou deux minutes avant de commencer à programmer, plutôt que de se reporter à l'annexe clavier trop souvent

- Arthur Charguéraud -