Chapitre 2

Langages de Programmation

Sommaire.

1 Caractériser un langage de programmation **1**
1.1 Programmation et pensée : les paradigmes. . . . 1
1.2 Exécution d'un programme, compilation, interprétation. . . . 2
1.3 Programmation et données : typage. . . . 2
1.4 Mots-clés et indentifiants. . . . 2

2 Le langage C. **2**
2.1 Les bases. . . . 2
2.1.1 Un premier programme en C. . . . 3
2.2 Types de base. . . . 3
2.2.1 Les entiers. . . . 3
2.2.2 Les flottants. . . . 3
2.2.3 Cohabitation entre types. . . . 3
2.2.4 Booléens. . . . 4
2.2.5 Chaînes de caractères. . . . 4
2.3 Les variables. . . . 4
2.3.1 Les identifiants. . . . 4
2.3.2 Syntaxe. . . . 4
2.3.3 Constantes. . . . 5
2.4 Les fonctions. . . . 5
2.4.1 Syntaxe. . . . 5
2.4.2 Typage. . . . 5
2.5 Structures de contrôle. . . . 5
2.6 Les pointeurs. . . . 6
2.6.1 Demandes de mémoire. . . . 6
2.6.2 Tableaux et pointeurs. . . . 6
2.7 Structures. . . . 7
2.7.1 Utilisation de base. . . . 8
2.7.2 Structures et pointeurs. . . . 8

3 Le langage OCaml. **8**
3.1 Un langage fonctionnel. . . . 8
3.2 Types. . . . 9
3.3 Expressions. . . . 9
3.3.1 Expressions conditionnelles. . . . 9
3.3.2 Expressions let. . . . 9
3.4 Fonctions. . . . 10
3.5 Listes et filtrages. . . . 10
3.6 Types algébriques. . . . 11

Les propositions marquées de ★ sont au programme de colles.

1 Caractériser un langage de programmation

1.1 Programmation et pensée : les paradigmes.

Définition 1: Paradigmes.

Les **paradigmes** permettent de caractériser la vision du déroulement d'un programme qu'a le développeur.
—Le paradigme impératif se caractérise par une suite d'instructions permettant de modifier la mémoire des processus par affectations successives.
—Le paradigme déclaratif fonctionnel se caractérise par des déclarations de données et de règles de déductions (fonctions mathématiques), les appliquant pour arriver à une solution.

Définition 2: Flot d'exécution.

Le **flot d'exécution** d'un programme impératif est l'ordre dans lequel les instructions de ce programme sont exécutées.

Définition 3: Graphe de flot de contrôle.

Le **graphe de flot de contrôle** d'un programme est l'ensemble de ses flots de contrôle donnés sous la forme d'un graphe.

Le contrôle explicite de ce flot par le programmeur se fait via des structures de contrôle (conditionnelles, boucles).

Définition 4: Graphe.

Un graphe est un objet mathématique qui contient deux types de données:
- Des sommets.
- Des arcs.

On peut suivre une suite d'arcs: on obtient un chemin.

1.2 Exécution d'un programme, compilation, interprétation.

Définition 5: Compilé, interprété.

Les langages **compilés** traduisent le code source en éxécutable.
Les langages **interprétés** lisent le code source et l'exécutent directement.

1.3 Programmation et données : typage.

Définition 6: Types

Les programmes traitent et manipulent des données.
Comme en maths, on peut regrouper ces données par ensembles de valeurs et d'opérations qui s'y appliquent :
- $\mathbb{N} : +,\ \times,\ /,\ \%$.
- $\mathbb{Z} : +,\ \times,\ -,\ /,\ \%$.
- $\mathcal{F}(\mathbb{R}, \mathbb{R}) : \circ$, l'ensemble des fonctions de $\mathbb{R}$ dans $\mathbb{R}$.

Ces groupes sont appelés des **types**.

Définition 7: Taille d'un type.

La **taille** d'un type est la place mémoire occupée par l'une de ses valeurs. Il existe des types à taille fixe et à taille variable. Les tailles sont exprimées soit en bits, soit en octets.

Chaque langage de programmation possède ses types de base et permet de définir de nouveaux types. Le typage est le fait d'attribuer un type à une valeur.

Définition 8: Conversions.

Dans certains langages de programmation, on peut sous certaines conditions convertir une valeur d'un type en un autre. Cette opération s'appelle une **conversion** ou un cast. Elle est **implicite** si le programmeur n'a pas besoin de la spécifier, sinon elle est **explicite**.

Exemple: En C, `3 + 2.5` entraîne la conversion implicite de `3` en flottant, ce qui est interdit en OCaml.

Définition 9: Typage faible, fort.

Plus il y a de conversions implicites dans un langage, plus on dit que son typage est **faible**, à l'inverse, le typage est dit **fort**.
Un typage fort permet d'éviter les erreurs de type très tôt dans le développement.

Ainsi, le C a un typage faible, et l'OCaml un typage fort.

Définition 10: Typage statique, dynamique.

Un langage possède un typage **statique** si les vérifications des types sont faites au moment de la compilation..
Un langage possède un typage **dynamique** si les vérifications des types sont faites au moment de l'exécution.

1.4 Mots-clés et indentifiants.

Définition 11: Mot-clé.

Les langages de programmation textuels possèdent tous des **mots-clés**. Ce sont des mots réservés par le langage, qui permettent de spécifier les programmes.
Par ailleurs, la plupart des langages permettent d'attribuer des noms à des données pour les réutiliser.

Définition 12: Identifiant.

Un **identifiant** est un nom qui permet de désigner une entité du langage.

2 Le langage C.

2.1 Les bases.

Définition 13: Le langage C.

Le langage C est compilé et possède un typage faible et statique.
Les fichiers source en C possèdent l'extension `.c`.
On les compile en ligne de commande avec la commande **gcc**.
On peut spécifier le nom de l'exécutable au moment de la compilation avec l'option `-o`.

Exemple: `gcc -o executable source.c`

2.1.1 Un premier programme en C.

Définition 14: Programme.

Un programme en C est un texte ou un ensemble de textes qui vérifient un certain nombre de règles. L'ensemble de ces règles s'appellent une grammaire, elles permettent d'obtenir un exécutable à la compilation.
La compilation peut donner lieu à divers messages d'erreurs :
— Des erreurs qui empêchent la fabrication de l'exécutable.
— Des warnings qui n'empêchent pas la fabrication, mais qui sont à analyser soigneusement.
Pour afficher tous les warnings, on utilise l'option `-Wall`.

Le compilateur détecte les erreurs liées à la syntaxe et aux types, mais il ne peut pas détecter les erreurs liées aux valeurs, comme une division par zéro.

Définition 15: La fonction main

Le flot de contrôle d'un programme C est l'exécution de sa fonction `main`.

La fonction `main` renvoie nécessairement un entier, de type `int`. Son exécution est linéaire, et peut s'arrêter avant la dernière ligne :
— Erreur à l'exécution.
— Mot-clé `return`.
— Appel à une fonction `exit()`.
Chaque instruction se termine par un point-virgule, et les blocs d'instructions sont délimités par des accolades.

2.2 Types de base.

2.2.1 Les entiers.

Définition 16: Entiers.

Il existe plusieurs types d'**entiers** en C:
— Des types dont on sait comparer les tailles, sans que la norme ne les fixe : `int`, `unsigned int`.
— Des types dont la taille est fixée par la norme : `int8_t`, `uint_8t`, `int32_t`, ...

Aucun type entier ne permet de représenter tout $\mathbb{N}$ ou $\mathbb{Z}$.

Définition 17: Entiers signés.

Ce sont des types qui peuvent représenter des entiers positifs et négatifs de façon symétrique.
— `int8_t`: 2^8 valeurs représentées dans $[\![-2^7, 2^7 - 1]\!]$.

Pour représenter les entiers positifs le premier bit est à 0, puis on a l'écriture du nombre en base 2 avec le bit de poids fort à gauche.

Définition 18: Complément à 2.

Le complément à 2 d'un nombre k codé sur n bits en base 2 est son complément à 2^n en base 2.
C'est l'écriture de $2^n - k$ en base 2 sur n bits, il sert à coder $-k$.

Exemple: Le complément à 2 de 37 sur 8 bits est 11011011.

Définition 19: Opérations entre entiers.

Comparaisons: `==`, `!=`, `<`, `>`, `<=`, `>=`.
Opérations: `+`, `-`, `*`, `/`, `%`.

Sur des opérations de même priorité, la priorité se fait de gauche à droite.

2.2.2 Les flottants.

Définition 20: Flottants.

Les réels sont représentés en C par des nombres à virgule flottante.
Le seul type au programme de MP2I/MPI est le `double`, représenté en général sur 8 octets.

Définition 21: Opérations entre flottants.

Comparaisons: `==`, `!=`, `<`, `>`, `<=`, `>=`.
Opérations: `+`, `-`, `*`, `/`.

2.2.3 Cohabitation entre types.

Définition 22

Une opération entre un entier et un double entraîne une conversion implicite de l'entier en double. Une somme entre deux entiers signés entraîne en général la conversion de l'entier du type le plus petit vers l'autre.
On évite en général les opération entre entiers signés et non signés.

Attention: $1/2$ donne 0, mais $1.0/2$ donne 0.5 en C.

2.2.4 Booléens.

Définition 23: Booléens.

Les booléns n'existent pas par défaut en C. Pour les utiliser, il faut inclure le fichier d'en-tête `stdbool.h`.
Les booléns sont représentés par le type `bool`, codé sur un octet. Il possède deux valeurs : `true` et `false`.
Ce type supporte trois opérations :

- `!` : la négation.
- `&&` : **et** logique.
- `||` : **ou** logique.

L'opérateur `!` est prioritaire sur les autres.
Les opérateurs `&&` et `||` sont paresseux : ils n'évaluent que le strict nécessaire, de gauche à droite.

Exemple: (`true || (1/0)`) ne provoque pas de division par zéro.

2.2.5 Chaînes de caractères.

cf. le chapitre sur les chaînes de caractères.

2.3 Les variables.

Définition 24: Variable.

En C, une **variable** est une boîte dans laquelle on met une valeur.
Une variable possède 4 caractéristiques ne pouvant pas être modifiées à l'exécution:

- Son nom.
- Son type.
- La façon dont les données sont interpretées.
- Son adresse.

En revanche, la valeur de la variable peut être modifiée pendant l'exécution.

2.3.1 Les identifiants.

Définition 25: Règles.

Les règles **strictes** de fabrication d'un identifiant en C sont les suivantes:

- Il commence par une lettre ou un underscore.
- Il ne contient que des lettres, des chiffres ou des underscores.
- Il ne peut pas être un mot-clé.

Les règles de **bonne partique** sont les suivantes:

- Éviter de prendre des indentifiants des librairies standard.
- Les identifiants doivent être uniques.
- Les identifiants doivent être intelligibles.
- Éviter les underscore en premier caractère.

2.3.2 Syntaxe.

Définition 26: Déclaration.

Avant de pouvoir utiliser une variable en C, il faut la **déclarer**: annoncer qu'on va l'utiliser en spécifiant son nom et son type.
Exemple: `int a;`

Suite à cette déclaration, le système réserve en mémoire une boîte de taille `sizeof(int)` à laquelle on peut accéder via le nom a.
On peut déclarer plusieurs variables de même type en une seule ligne : `int a, b, c;`.

Définition 27: Affectation.

Pour donner une valeur à une variable, ou la modifier, on utilise le nom de la variable suivi de l'opérateur d'**affectation** `=`, puis de l'expression qui donne sa valeur.
Exemple: `int a; a = 3;`

Il faut **toujours** initialiser les valeurs.

Définition 28: Initialisation.

La première affectation s'appelle l'**initialisation** de la variable.
On peut déclarer et initialiser une variable sur la même ligne : `double pi = 3.0;`.

Définition 29: Portée.

Une variable n'est pas visible avant d'être déclarée, elle cesse d'exister à la fin du bloc dans lequel elle est déclarée: l'ensemble du code dans lequel la variable est utilisable s'appelle sa **portée**.

On ne peut pas déclarer deux variables de même nom dans le même bloc, mais on peut réutiliser un même identifiant pour une variable dans un sous-bloc. Dans ce cas, la variable du sous-bloc masque la variable externe. Ce masque n'est plus accessible en sortie du sous-bloc.

Pour maintenir la lisibilité du code, on évite de masquer des variables.

2.3.3 Constantes.

Définition 30: Constantes.

En C, les **constantes** sont des valeurs qu'on ne peut pas modifier: —Les constantes littérales: constantes numériques ou chaînes écrites par le programmeur pour lesquelles l'allocation mémoire et l'initialisation se font à la compilation.
— Les constantes symboliques: une variable dont on ne peut pas modifier la valeur.

Exemple: `const int maxint = 2147483647;`

2.4 Les fonctions.

2.4.1 Syntaxe.

Définition 31: Fonction.

Une **fonction** est une boîte noire à laquelle on transmet des valeurs et qui soit renvoie un résultat, soit modifie l'état du système soit produit un affichage, soit effectue une combinaison de ces actions.

L'exécution d'une fonction dépend des valeurs transmises et de l'état du système.

Définition 32: Appel.

On peut exécuter la suite d'actions décrite dans une fonction autant de fois qu'on le souhaite, en utilisant le nom de la fonction suivi de parenthèse et de valeurs pour ses paramètres.
On dit qu'on invoque (on appelle) la fonction.
Les valeurs transmises à une fonction peuvent être littérales ou être le résultat d'un calcul.

Exemple: $\sin(3)$, $\sin(3 * pi/2)$.

La syntaxe pour écrire une fonction en C est la suivante:

```
type_de_retour nom(type nom1, type nom2, ...) { code }
```

Le type de retour peut être un type quelconque ou fabriqué. Si la fonction ne renvoie rien, on utilise `void`.
Le nom d'une fonction doit être un identifiant admissible et non encore utilisé.
Il est possible d'écrire des fonctions sans paramètres, il faut quand même mettre des parenthèses.

Exemple: `int somme(int a, int b) { return a+b; }`

2.4.2 Typage.

Définition 33: Signature.

La **signature** d'une fonction contient une liste de couples (type, nom) qui sont les paramètres de la fonction.
Ils permettent d'identifier les valeurs dont la fonction a besoin pour faire son travail.
En C, les paramètres sont transmis par valeurs, la fonction travaille sur une copie des valeurs qui sont transmises.

Définition 34: Valeur de retour.

Si le type de retour annoncé n'est pas `void`, alors la fonction doit contenir un retour pour tout chemin du graphe de flot de contrôle, le type de retour doit être compatible avec celui annoncé.

2.5 Structures de contrôle.

Définition 35: Structure de contrôle.

Les **structures de contrôle** sont des instructions du langage qui permettent au programmeur de contrôler le chemin du graphe de flot de contrôle.

Définition 36: Conditions

Les **conditions** permettent de choisir entre plusieurs chemins d'exécution.
En C, on utilise la structure `if` pour cela:

```
if (condition) { instructions } else { instructions }
```

Définition 37: Boucle conditionnelle.

Les **boucles conditionnelles** permettent de répéter un bloc d'instructions tant qu'une condition est vraie.
En C, on utilise la structure `while` pour cela:

```
while (condition) { instructions }
```

Définition 38: Boucle inconditionnelle.

es **boucles inconditionnelles** permettent de répéter un bloc d'instructions un nombre de fois donné.
En C, on utilise la structure `for` pour cela:

```
for (initialisation; condition; incrémentation) { instructions }
```

2.6 Les pointeurs.

Définition 39: Pointeur.

Un **pointeur** est une adresse dans la mémoire d'un processus.
Le type de l'adresse d'une variable de type `t` est `t*`

Exemple: `int* p;` est un pointeur sur entier.

La taille d'un type pointeur est de 8 octets. Pour affecter une valeur à un pointeur, il faut pouvoir écrire une expression qui s'évalue comme une adresse:
— Prendre l'adresse d'une variable existante du bon type.
— Demander au système une adresse libre de la bonne taille.
— Invoquer une fonction qui renvoie un type pointeur.
— Utiliser une expression littérale constante de type pointeur.

Il n'y a qu'une contante littérale de type pointeur, définie dans `stdlib` et de nom `NULL`. Le standard du C garantit qu'elle est différente de toute adresse utilisable.

Définition 40: Adresse.

L'**adresse** d'une variable s'obtient en écrivant `&` devant le nom de la variable.

Définition 41: Valeur, contenu.

La **valeur** d'une variable de type pointeur sur `t` est une adresse. Son **contenu** est la valeur de la case sur laquelle elle pointe, de type `t`.

Définition 42: Déréférencement.

Pour accéder au contenu d'un pointeur, on écrit le caractère `*` devant le nom du pointeur: on le **déréférence**.
Il ne faut jamais déréférencer `NULL`.

2.6.1 Demandes de mémoire.

Définition 43: Allocation.

On peut demander au système l'adresse d'une portion de mémoire à laquelle on aurait accès avec:

```
void* malloc(size_t size);
```

Le type `void*` peut être implicitement converti en tout type de pointeur.
Si `malloc` n'arrive pas à allouer de mémoire, il renvoie `NULL`.
Il n'y a pas de moyen de connaître la taille d'une zone allouée, il faut donc utiliser les méthodes usuelles.
On accède au contenu de la $i^{\text{ème}}$ d'un pointeur `p` case en écrivant `p[i]`.

Définition 44: Libération.

En C, le programmeur doit signaler lui-même au système quel emplacement mémoire alloué par `malloc` est libre et peut être réutilisé :

```
void free(void* p);
```

Il y a une conversion implicite de tout type de pointeur vers `void*`.
On ne peut libérer qu'en fournissant une adresse de début d'allocation.

2.6.2 Tableaux et pointeurs.

Définition 45: Contraintes.

Dans un monde idéal, la mémoire d'un processus aurait plein de propriétés intéressantes:
— Accès immédiat à n'importe quelle case.
— Obtenir immédiatement un emplacement mémoire.
— Libérer à n'importe quel moment un emplacement mémoire.
— Modifier la taille d'un emplacement mémoire.

Comme il n'existe pas de structure de donnée fournissant tous ces services, on les sépare.

Définition 46: Piles.

Une **pile** est une structure de type LIFO (Last In First Out), on ne peut accéder qu'à la dernière case ajoutée.
Opérations: `empiler`, `dépiler`, `est_vide`, `sommet`.

Définition 47: La pile.

La pile ne contient pas que des couples identifiants-valeurs, mais également les noms des fonctions appelées. Ce sont les appels qui sont empilés.

À chaque invocation de fonction, un bloc de données est créé, et empilé sur la pile du processus, c'est le bloc d'activation de l'appel (stack frame).
C'est dans ce bloc que sont écrites toutes les données locales à l'appel de la fonction, au retour de l'appel, le bloc est effacé de la pile. L'empilement et le dépilement de blocs se font rapidement car la structure de pile est faite pour ça.

Le bloc d'activation contient en particuler:
— Les paramètres de la fonction initialisés avec les valeurs des arguments.
— Les variables et tableaux locaux.
— Un emplacement pour la valeur de retour affecté avec l'évaluation de l'expression qui suit le retour.
— L'adresse de l'instruction à exéctuer après le retour.

La gestion de la pile d'appels incombe entièrement au système d'exploitation.
Les information locales à un appel sont perdues à la sortie de l'appel, la fonction appelante ne récupère que la valeur de retour.

Définition 48: Le tas.

Parfois, on a besoin qu des informations survivent à un appel. La meilleure solution est alors de les stocker dans le tas. Si on alloue de la mémoire dans une fonction et qu'on renvoie l'adresse de l'emplacement mémoire, la fonction appelante peut récupérer la valeur de retour et accéder à l'emplacement mémoire.

Les paramètres d'une fonction peuvent être de type pointeur, la fonction recopie alors l'adresse, ce qui lui permet d'en modifier le contenu. À la sortie de l'appel, le bloc d'activation disparaît donc cette copie aussi, mais son contenu ne disparaît pas car stocké dans le tas, ou dans un autre bloc.

Définition 49: Tableaux.

Un **tableau** est une suite d'éléments de même type, stockés en mémoire de façon contigue.
En C, un tableau est un pointeur sur la première case du tableau.
Pour accéder à la $i^{\text{ème}}$ case d'un tableau, on écrit `t[i]`.

Exemple: `int t[5]; t[0] = 3; t[1] = 4; t[2] = t[0] + t[1];`

Définition 50: Tableaux multidimensionnels.

Les **tableaux multidimensionnels** dont les données sont déterminées statiquement sont rangés en mémoire comme un tableau unidimensionnel, dans lequel on découpe chaque ligne. Au contraire, les tableaux dont les données sont déterminées dynamiquement ne représentent pas une zone mémoire contigue.

Exemple: `int t[2][3]; t[0][0] = 3; t[1][2] = 4;`

Définition 51: Conséquence sur les fonctions.

Bien entendu, on ne peut pas renvoyer un tableau statique. Une fonction ne peut traiter de la même manière un emplacement mémoire à découper et un ensemble d'emplacements mémoire disjoints.

Définition 52: Arguments du main.

Pour donner un argument à une command dans le shell, on l'écrit après la command.
Les programmes en C récupères ces arguments comme valeur de paramètre du `main`.
Pour s'en servir, on écrit `int main(int argc, char* argv[])`.
Au lancement du programme, les paramètres `argc` et `argv` reçoivent:
— `argc` : le nombre d'arguments passés au programme.
— `argv` : un tableau de chaînes de caractères contenant les arguments passés au programme.

Exemple: `./a.out 3 4` donne `argc = 3` et `argv = ["./a.out", "3", "4"]`.

2.7 Structures.

Définition 53: Structure.

Une **structure** agrège plusieurs informations qui peuvent avoir le même type ou non.
Chaque information de la structure s'appelle un **champ** (ou attribut).
En quelque sorte, les tableaux permettent d'agréger des données numérotées, alors que les structures permettent d'agréger des données nommées.

2.7.1 Utilisation de base.

Définition 54: Syntaxe.

La création d'un type structuré se fait grâce au spécifiacteur `struct` suivi du nom du type.

```
struct point { double x; double y; } ;
```

Cette instruction crée le type `struct point` qui contient deux champs de type `double` nommés `x` et `y`.
Une structure peut **s'auto-référencer**, c'est-à-dire contenir un champ de type pointeur sur elle-même.

Définition 55

On peut initialiser une variable de type structuré grâce à un initialisateur:

```
struct point p = {.x = 3.0, .y = 4.0};
```

Pour accéder à un champ d'une telle variable, on écrit le nom de la variable, un point et le nom du champ.

Exemple: `p.x == 3.0;` est vrai.

2.7.2 Structures et pointeurs.

Définition 56

On peut considérer l'adresse d'une variable de type structuré, ou utiliser un champ de type structure.
Attention: l'opérateur `.` est prioritaire sur `*`.
On utilise l'opérateur `->` pour accéder à un champ d'une structure pointée.

Exemple: `struct point* p; p->x = 3.0;` pour une structure pointée.
Exemple: `struct point p; p.x = 3.0;` pour une structure non pointée.

Si on a besoin de modifier la valeur d'une structure, on transmet son adresse.
Si une structure est grande, on transmet son adresse.

Exemple 57: Listes chaînées.

Un exemple de structure auto-référencée est la liste chaînée.
C'est une structure qu'on peut définir récursivement:
— Soit c'est une liste vide.
— Soit elle contient une valeur et une référence vers une autre liste chaînée.

3 Le langage OCaml.

3.1 Un langage fonctionnel.

Définition 58: Langage fonctionnel.

OCaml est un **langage fonctionnel**: l'abstraction de base est la fonction au sens mathématique du terme. Une fonction associe une sortie à une entrée.

Définition 59: Mutabilité.

Pour tout le début du cours, on s'astreint à n'utiliser que la partie immuable de OCaml, on ne peut pas modifier la valeur d'un emplacement mémoire.
En C, l'état du programme est mutable: on change en permanence les valeurs des variables.

Dans un programme purement fonctionnel, les variables ne changent pas de valeur, et les fonctions n'ont pas d'effet de bord.

Définition 60: Expressions.

En programmation fonctionnelle, il n'y a pas d'instructions mais des **expressions**: elles ne décrivent pas comment faire le calcul, mais ce qu'il faut calculer.
Les expressions sont la brique de base des langages fonctionnels. Chaque expression a deux facettes qui nous permettent de l'écrire:
— Sa syntaxe: mots-clés, ponctuation.
— Sa sémantique: signification.
Il y a deux types de sémantiques:
— Statique (avant l'exécution du programme), le compilateur produit un type pour chaque expression ou échoue.
— Dynamique (règles d'évaluation), production d'une valeur si tout se passe bien, une valeur est une expression qui n'a plus besoin d'être évaluée.

Dans le cas où l'évaluation échoue, on se retrouve avec une erreur ou une boucle infinie.

Définition 61: Exécuter du OCaml.

Pour obtenir du bytecode, on utilise la commande `ocamlc source.ml`.
Elle produit un fichier `a.out` qu'on peut exécuter avec `ocamlrun a.out`.
Pour obtenir un exécutable natif, on utilise la commande `ocamlopt source.ml`.

3.2 Types.

Définition 62: Types de base.

OCaml possède des types de base:
— `int` : entiers `(+, -, *, /, mod)` codé sur 63 bits.
— `float` : flottants `(+., -., *., /.)` en norme IEE.
— `bool` : booléens `(&&, ||, not)`.
— `char` : caractères codé sur 8 bits en ASCII.
— `string` : chaînes de caractères `(^)`.

Définition 63: Comparaisons.

On peut comparer les entiers, les flottants, les booléens et les caractères.
Les comparaisons se font avec `=`, `<>`, `<`, `>`, `<=`, `>=`.
On ne compare pas de types différents.

Définition 64: Conversions.

OCaml ne fait pas de conversions implicites.
Pour convertir un entier en flottant, on utilise `float_of_int`.
Pour convertir un flottant en chaîne de caractères, on utilise `string_of_float`.
Pour convertir un flottant en entier, on utilise `int_of_float`.
...

Définition 65: Inférences de types.

Le compilateur infère les types: le programmeur ne spécifie pas les types.
Si le compilateur ne parvient pas à les inférer, la compilation échoue.
On peut spécifier les types avec `let (nom : type) = expression`.

Définition 66: Définitions des variables.

Une variable permet de donner un nom à une valeur. En OCaml, cette valeur ne pourra pas être modifiée.
Pour définir une variable et lui donner une valeur, on utilise le mot-clé `let`:

```
let nom = expression
```

où nom est un identifiant. Les règles de construction d'un identifiant sont globaleme,nt les mêmes qu'en C. L'identifiant commence nécessairement par une minuscule.

Une définition n'est pas une expression.

3.3 Expressions.

3.3.1 Expressions conditionnelles.

Définition 67

En OCaml, on utilise le mot-clé `if` pour les expressions conditionnelles.
La syntaxe est la suivante:

```
if condition then expression_1 else expression_2
```

L'expression conditionnelle renvoie la valeur de `expression_1` si la condition est vraie, et celle de `expression_2` sinon.
La condition doit être de type booléen et les deux expressions doivent être du même type.

3.3.2 Expressions let.

Définition 68

Les expressions let ressemblent syntaxiquement aux définitions mais sont des expressions qu'on peut utiliser comme sous-expressions d'une expression plus grande:

```
let nom = expression_1 in expression_2
```

Sémantique dynamique:
— On évalue `expression_1` en `valeur_1`.
— On remplace `nom` par `valeur_1` dans `expression_2`.
— On évalue `expression_2` en `valeur_2`.
— On renvoie `valeur_2`.

Sémantique statique:
— On vérifie que `expression_1` est bien de type `t_1`.
— Si `nom` est de type `t_1`, alors `expression_2` est de type `t_2`.
Alors le type de l'expression est `t_2`.

Chaque `let` définit une nouvelle variable. Si elle porte le même nom qu'une variable existante, elle la masque, l'ancienne variable existe toujours.

3.4 Fonctions.

Définition 69: Fonctions anonymes.

En OCaml, on peut définir des fonctions de manière similaire aux maths:

```
fun x -> x + 1
```

On utilise le mot-clé `fun`.
Pour appliquer une fonction à une valeur, on écrit la fonction, un espace puis la valeur.

Définition 70: Fonctions.

La syntaxe précédente est un peu lourde, OCaml possède une autre syntaxe pour définir et donner directement un nom à une fonction : `let suivant x = x + 1`
Cette expression est sémantiquement équivalente à la précédente, mais possède une autre syntaxe. La deuxième écriture est un sucre syntaxique.

Définition 71: Sémantique statique.

Le type d'une fonction contient une flèche : $t \to u$ signifie que la fonction possède un unique paramètre de type t et que son évaluation est de type u.

Si une fonction possède deux arguments, son type est : $t_1 \to t_2 \to u$.
L'application d'une fonction est prioritaire sur les opérateurs arithmétiques.

Définition 72: Sémantique dynamique.

Chaque expression en argument est évaluée puis la fonction est évaluée sur les valeurs.

Définition 73: Curryfication.

On a vu que dans le type d'une fonction, le symbole flèche semble avoir deux significations différentes, en fait, ce n'est pas le cas: on ne définit que des fonctions à un seul argument en OCaml.
La notation `fun x y → e` est un sucre syntaxique pour `fun x → (fun y → e)`.
Dans cette notation, on voit que l'argument est unique.

L'associativité de → se fait de gauche à droite, donc `x → (y → e)` peut s'écrire `x → y → e`.

Définition 74: Fonction récursives.

Il faut spécifier explicitement qu'une fonction est récursive avec le mot-clé `rec`.
Pour définir des fonctions mutuellement récursives, on utilise le mot-clé `and`.
Exemples:
— `let rec f x = if x = 0 then 1 else x * f (x-1)`.
— `let rec f x = if x = 0 then 1 else x * g (x-1) and g x = if x = 0 then 1 else x * f (x-1)`.

Définition 75: Récursion terminale.

OCaml possède un mécanisme qui lui permet de remplacer un bloc d'appel par un autre si le résultat de l'appel est en fait le résultat de l'appel suivant.
Une fonction récursive qui ne fait pas d'opérations sur le résultat de son appel récursif est dite récursive terminale.
Les appels ne s'empilent pas donc la pile d'appel ne déborde pas.

3.5 Listes et filtrages.

Définition 76: Polymorphisme.

Les identifiants de types commencent toujours par une apostrophe, ils sont appelés **variables de types**.
Lorsqu'une fonction est **polymorphe**, elle peut être appliquée à des arguments de types différents.

Exemple: `let fst (x, y) = x` est de type `'a * 'b -> 'a`.

Définition 77: Listes.

Les **listes** sont des listes chaînées. Elles s'écrivent entre crochets et ses élément sont séparés par des points-virgules. Cette syntaxe est un sucre syntaxique pour : `t::s` où t est la tête de liste et s la suite. Les listes sont immuables.

Exemple: `[1; 2; 3]` est une liste d'entiers.

Sémantique dynamique:
— `[]` est une valeur.
— si `e` s'évalue en `v` et `e'` s'évalue en la liste `l`, alors `e::e'` s'évalue en la liste `v::l`.

Sémantique statique:
Pour un type `t`, le type `t list` est le type des listes dont les cases sont de type `t`.
— `[]` est de type `'a list`.
— si `e` est de type `t` et `e'` est de type `t list` ou `'a list`, alors `e::e'` est de type `t list`.

Définition 78: `@` vs `::`

L'opérateur `@` permet de concaténer deux listes: `l1 @ l2`.
La complexité de l'opérateur `@` est linéaire en le nombre d'éléments dans la liste, alors que celle de `::` est constante.

Définition 79: Filtrage par motif.

Cette technique permet d'appliquer un traitement à une liste selon ses premiers maillons:

```
match liste with
| [] -> e1
| t::q -> e2
```

Cette technique compare la valeur d'une expression à plusieurs motifs. Le premier identifiable à cette valeur est choisi et l'expression associée est évaluée.

Les retours à la ligne dans les `match` sont facultatifs, comme la barre avant le premier motif.

Les motifs peuvent utiliser des identifiants, des jokers (`_`) et des constantes. Il y a des contraintes sur les variables de motifs:
— On ne peut pas utiliser deux fois la même varibale dans un même motif : `x::x::x::_` est illégal.
— On ne peut pas utiliser une variable existante dans un motif (la masque).

Définition 80: Fonctionnement du filtrage.

— Une constante s'identifie uniquement avec elle-même.
— Une variable s'identifie avec toute valeur du même type qui lui est associée.
— Le caractère `_` s'identifie avec n'importe quelle valeur.
— Chaque type de données a ses propres motifs de filtrages.

Sémantique dynamique:
Pour un filtrage de motif:

```
match e with
| m1 -> e1
| m2 -> e2
| ⋮
| mn -> en
```

— `e` est évalué en `v`.
— Si `v` s'identifie avec le premier motif possible.
— L'expression associée est évaluée en substituant aux variables de motif les valeurs associées.
— La valeur obtenue est la valeur de l'expression.

Sémantique statique:
Les motifs `mi` doivent être compatibles avec le type de `e` et les expressions `ei` doivent être de même type.

Remarque: Les motifs peuvent être imbriqués.

Définition 81: Disjonction de motifs.

On peut regrouper plusieurs motifs qui sont associés à la même expression: `| m1 | m2 | ... | mn -> e`.
Ces motifs doivent utiliser exactement les mêmes variables de motif, quel que soit le motif identifié à l'expression.

Définition 82: Motifs gardés.

On peut ajouter une condition à un motif: `| m when c -> e`.
La condition `c` est une expression booléenne qui doit être vraie pour que le motif soit identifié.

Définition 83: Mot-clé `function`

On filtre le dernier argument fourni à une fonction grâce à un sucre syntaxique: `function`.
Exemple: `let f = function | [] -> 0 | x::q -> x`.
Cet exemple est équivalent à `let f x = match x with | [] -> 0 | x::q -> x`.

3.6 Types algébriques.

Définition 84: Types énumérés.

Un **type énuméré** (ou somme) représente un ensemble fini de valeurs.
En OCaml, on utilise le mot-clé `type` pour définir un type énuméré.
Le nom d'un type commence par une lettre miniscule, chaque valeur particulière d'un type énuméré s'appelle un constructeur et son identifiant commence par une majuscule. Dans la déclaration d'un type somme, les noms des constructeurs sont séparés par des barres verticales.
On peut évaleur une expression de type énuméré par filtrage.
Exemple: `type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche`.
Sémantique dynamique:
Un constructeur est constant.
Sémantique statique:
Si `t` est défini par `type t = c1 | c2 | ... | cn`, alors `ci` est de type `t`.
En cas d'ambiguïté sur un constructeur, le dernier type déclaré dans lequel il apparaît est choisi.

Définition 85: Types énumérés avec données.

Les types énumérés peuvent être enrichis avec des valeurs.
Syntaxe: `type t = c1 of t1 | c2 of t2 | ... | cn of tn`.
Un constructeur qui ne porte pas de valeur est constant, pour écrire une expression de type `t` avec un tel constructeur on utilise son nom.
Un constructeur qui transporte de l'information est non constant, on utilise alors le nom du constructeur suivi de l'information qu'il transporte.

Exemple: `type nombre = Int of int | Float of float`.
On peut alors écrire `Int 3` ou `Float 3.0`.

Définition 86: Types récursifs et polymorphismes.

En OCaml, on peut définir des types récursifs, on peut définir des types polumorphes en leur ajoutant un paramètre sous la forme de variable de type avant son nom.
Exemple: `type 'a liste = Vide | Maillon of 'a * 'a liste`.

On peut utiliser plusieurs paramètres de type.
Exemple: `type ('a, 'b) couple = 'a * 'b`.

Autre exemple de liste chaînée:
```
type maillon {valeur : int; suivant : liste}
and liste = Vide | Maillon of maillon.
```

Définition 87: Enregistrements.

Un **enregistrement** est un type qui permet d'agréger des données (dont le type existe).
Il y a un champ par type, et il faut donner une valeur à chaque champs, sinon le compilateur renvoie une erreur.
Pour récurpérer la valeur d'un champ, on utilise le nom de la variable point le nom du champ.

Exemples:
Définition du type: `type point = {x : float; y : float}`
Déclaration d'une instance de ce type: `let p = {x = 3.0; y = 4.0}`
Accès au champ `x` de `p`: `p.x`

On peut faire du filtrage par motif en ajoutant le motif : `{f1 = v1; f2 = v2; ...}`.
Où les `fi` sont des identifiants de champs et les `vi` des motifs compatibles avec le type du champ.

Définition 88: n-uplets

Un **n-uplet** est un type qui permet d'agréger des données non-nommées.
Une valeur de type n-uplet s'écrit `(e1,...,en)`, et son type est `t1 * t2 * ... * tn` où les `ti` sont les types des éléments du n-uplet.
On peut récupérer les valeurs d'un n-uplet:

```
let v1, v2, ... = (e1, e2, ...)
```

Quitte à utiliser un joker pour les valeurs qui ne nous intéressent pas.
On ajoute le motif `(m1, m2, ...)` dans les motifs autorisés, avec parenthèses facultative, mais le motif doit être de même taille que le n-uplet filtré.

Définition 89: Type option.

Le type `option` permet de gérer les valeurs qui peuvent être absentes.
Définition du type: `type 'a option = None | Some of 'a`.
C'est un type présent dans la bibliothèque standard d'OCaml.