2 Leçon 2

2.1 Langages impératifs, le langage C

Aujourd'hui nous examinons plus en détail les constructions de base d'un langage impératif typique : le langage C, inventé par Kernighan et Ritchie au début des années 1970.

La chose essentielle à retenir est qu'un programme C définit une *séquence* d'instructions élémentaires, qui sont des *affectations* : pour simplifier, les affectations sont des instructions de la forme ⟨variable⟩ `=` ⟨expression⟩`;`, qui ont pour effet de calculer la valeur de l'expression à droite du signe =, et de la ranger ensuite dans la variable à gauche du signe =. L'organisation des affectations en séquence se fait à l'aide d'une certain nombre de constructions du langage, que nous verrons en section 2.1.3. Cette présentation de C n'est pas complète, et on pourra consulter différentes ouvrages de référence, par exemple *Le langage C*, par Kernighan et Ritchie, ou la référence `http://diwww.epfl.ch/w3lsp/teaching/coursC/` par exemple. Pour les plus courageux, on pourra aussi consulter la norme ISO/IEC 9899-1999 définissant le C dit "ANSI C" (le C officiel).

2.1.1 Affectations

Par exemple,

```
x = 3;
```

a pour effet de mettre l'entier 3 dans la variable nommée `x`. Un autre exemple est l'instruction

```
x = y+1;
```

qui récupère l'entier stocké dans la variable `y`, lui ajoute 1, et range le résultat dans la variable `x`.

Attention ! Le symbole = n'est *pas* le prédicat d'égalité que l'on rencontre usuellement en mathématiques. La signification typique de $x = 3$ en mathématique serait celle d'une valeur de vérité, valant vrai si x vaut 3, et faux sinon. Ici, `x = 3` ne teste pas du tout si `x` égale 3 ou non. En revanche, il est vrai qu'une fois l'instruction `x = 3` exécutée, le contenu de la variable `x` vaut effectivement 3. Un moyen de voir à quel point = n'est pas le prédicat d'égalité des mathématiques est de considérer une instruction telle que :

```
x = x+1;
```

qui récupère l'entier stocké dans la variable `x`, lui ajoute 1, et range le résultat de nouveau dans `x`. Donc, par exemple, si `x` contient l'entier 3 avant d'effectuer `x = x+1;`, `x` contiendra 4 après. En particulier, en aucun cas on n'aura `x` égal à `x`+1 après exécution de cette instruction. Mais bien sûr la valeur de `x` après cette instruction égale la valeur de `x` avant.

Cette ambiguïté, qui peut surprendre ceux qui sont habitués aux mathématiques, a été levée dans d'autres langages impératifs, comme Pascal, où l'on écrirait `x := x + 1`, en utilisant le symbole `:=` pour bien marquer qu'il s'agit d'une affectation et non d'un test d'égalité.

Si = est l'affectation, le test d'égalité en C est le symbol ==. On écrira donc :

```
if (x==3) ...
```

pour tester si `x` vaut 3, et non

```
if (x=3) ...
```

qui, dans un langage un peu plus sûr que C, serait une erreur, et serait rejeté par le compilateur. Mais C l'accepte gaiement… le langage C est en fait très permissif : la sémantique de C précise en effet que toute affectation, par exemple `x=3`, peut aussi être vue comme une expression, et a pour valeur la valeur de son côté droit. Ici, la valeur de `x=3` est donc 3, ce qui permet d'écrire des expressions comme `y=x=3;`, qui range 3 dans `x` (`x=3`), puis retourne le résultat de l'affectation `x=3`, c'est-à-dire 3, et le range dans `y` : donc `y=x=3;` range 3 dans `x` et dans `y`.

La syntaxe des expressions de C est assez riche, et un aperçu (pas tout à fait complet, voir n'importe quel livre d'introduction à C pour l'ensemble complet) est donné en figure 3.

$$\begin{array}{rlll}
e & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{ma_variable} \mid \ldots & \text{variables}\\
 & \mid & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{-1} \mid \texttt{-2} \mid \ldots \mid \texttt{'a'} \mid \texttt{'b'} \mid & \\
 & & \quad \ldots \mid \texttt{3.1415926536} \mid \ldots & \text{constantes}\\
 & \mid & e+e \mid e-e \mid e*e \mid e/e \mid e\texttt{\%}e \mid -e & \text{opérations arithmétiques}\\
 & \mid & e\texttt{\&}e \mid e\texttt{|}e \mid e\texttt{\^{}}e \mid e\texttt{>>}e \mid e\texttt{<<}e \mid \texttt{\~{}}e & \text{opérations bit à bit}\\
 & \mid & e\texttt{==}e \mid e<e \mid e<=e \mid e>e \mid e>=e & \text{comparaisons}\\
 & \mid & e\texttt{\&\&}e \mid e\texttt{||}e \mid \texttt{!}e & \text{et, ou, non logiques}\\
 & \mid & *e \mid e[e] \mid \texttt{\&}e & \text{pointeurs, tableaux}\\
 & \mid & e(e,\ldots,e) & \text{appel de fonction}
\end{array}$$

FIG. 3 – Un aperçu de la syntaxe des expressions de C

On ne décrira pas la sémantique des différentes expressions de C. Sur les entiers (qui sont les entiers machines, pas tous les entiers ! sur une architecture de machine 32 bits standard, les entiers sont ceux compris entre $-2^{31} = -2\ 147\ 483\ 648$ et $2^{31} - 1 = 2\ 147\ 483\ 647$), $e_1 + e_2$ calcule l'addition des valeurs de e_1 et e_2 (modulo 2^{32} sur une architecture 32 bits ; pour être précis, la valeur de $e_1 + e_2$ sur ces machines vaut l'unique entier compris entre -2^{31} et $2^{31} - 1$ qui est congru à la somme des valeurs de e_1 et e_2 modulo 2^{32}) ; $e_1 - e_2$ calcule la différence de e_1 et e_2 (modulo 2^{32} de nouveau sur une architecture 32 bits), $e_1 * e_2$ calcule le produit de e_1 et e_2 (modulo 2^{32}), e_1/e_2 calcule leur quotient et $e_1\texttt{\%}e_2$ le reste de la division de e_1 par e_2 (modulo 2^{32}). Attention : e_1/e_2 ne calcule pas le résultat de la division de e_1 par e_2, mais sa partie entière (le quotient, donc). Par exemple, $2/3$ en C donne 0, pas $0.666666\ldots$; et `2%3` calcule le reste, ici 2.

En revanche, on peut aussi calculer en C sur des *nombres flottants* (abréviation de “nombres à virgule flottante”, qui dénote la façon de les représenter sur machine), qui sont des approximations finies de réels. Les opérations $+$, $-$, $*$ et $/$ sont alors les addition, soustraction, multiplication et division usuelles, à arrondi près. Par exemple, `2.0` 3.0/ donne vraiment `0.66666666666`. On remarquera que les nombres flottants sont écrits avec un point décimal ; en particulier, `2` et `2.0` sont deux objets différents : l'un est un entier machine, l'autre un flottant.

▸ **EXERCICE 2.1**
Que valent `1-2`, `15/7`, `15%7`, `1+(3*4)`, `(1+3)*4`, `1+3*4` en C ? Vous pouvez vous aider du compilateur `gcc`, en écrivant un programme de la forme :

```
#include <stdio.h>

int main ()
{
  printf ("La valeur de 1-2 est %d.\n", 1-2);
  return 0;
}
```

L'instruction `printf` ci-dessus a pour effet d'imprimer la valeur souhaitée : faire `man printf` pour avoir le modus operandi précis de `printf`.

Attention : ceci vous donnera la sémantique de `1-2` sur votre machine, pour votre compilateur particulier. N'en concluez pas nécessairement que la sémantique de C *en général* sera celle que vous aurez observée sur votre machine. Dans les exemples ci-dessus, cependant, les indications de votre machine seront fiables, et généralisables à toute machine (a priori).

▸ **EXERCICE 2.2**
Que valent `(-10)/7`, `(-10)%7` sur votre machine ? Ceci correspond-il à ce que vous attendiez ? Si q est le quotient et r le reste de la division de $a = -10$ par $b = 7$, a-t-on $a = bq + r$?

▸ **EXERCICE 2.3**
Que valent `10/(-7)`, `10%(-7)`, `(-10)/(-7)`, `(-10)%(-7)` sur votre machine ? Inférez-en la spécification du quotient et du reste sur votre machine : a/b et $a\%b$ calculent q et r tels que $a = bq + r$, $|r| < |b|$... et quelles autres propriétés déterminant q et r de façon unique ?

▸ **EXERCICE 2.4**
Il y a aussi toujours des cas pathologiques. Que valent `1/0`, `(-1)/0`, `0/0` sur votre machine ? (Attention, ceci est *totalement dépendant* de votre compilateur et de votre machine : sur la mienne, le programme est interrompu par un message `Floating point exception`, alors même qu'il ne s'agit pas de calcul en flottant mais bien en entier.)

▸ **EXERCICE 2.5**
(Si votre machine a des entiers 32 bits en complément à deux, ce qui est le cas pour toutes les machines de l'ENS à la date d'écriture de ce document.) Rappelez-vous que le plus petit entier représentable en machine est $-2^{31} = -2\ 147\ 483\ 648$, et le plus grand est $2^{31} - 1 = 2\ 147\ 483\ 647$. L'opposé du plus petit entier représentable n'est donc pas représentable ! En expérimentant avec `gcc`, comment calcule-t-il l'opposé du plus petit entier représentable ? Que valent `(-2 147 483 648)/(-1)`, `(-2 147 483 648)%(-1)` ?

2.1.2 Tableaux, structures

J'ai un peu menti en section 2.1.1 en disant que les affectations étaient de la forme $\langle$variable$\rangle$ =$\langle$expression$\rangle$ en C. Les côtés gauches peuvent en fait être plus généraux.

C'est notamment le cas avec les *tableaux*. En C, on peut déclarer un tableau, à l'entrée d'une fonction, par une déclaration de la forme

```
int a[50];
```

par exemple. Ceci déclare `a` comme étant un tableau de 50 éléments, chaque élément étant un entier machine (de type `int`). Alors que `int b` déclare `b` comme une variable contenant un seul entier, `a` ci-dessus en contient une rangée de cinquante.

On peut accéder à chaque élément du tableau en écrivant `a[`e`]`, où e est une expression retournant un entier. L'expression `a[0]` a pour valeur le premier élément du tableau, `a[1]` le deuxième, ..., `a[49]` le cinquantième (attention au décalage !). Les expressions `a[-1]`, `a[50]`, `a[314675]`, qui intuitivement dénoteraient des accès en-dehors du tableau, n'ont aucune sémantique : si l'on insiste pour obtenir leur valeur, on obtient en général n'importe quoi.

On peut aussi écrire des expressions d'accès aux éléments de tableaux plus compliqués. Par exemple, si `i` est une variable entière, `a[i]` dénote l'élément numéro `i` (qui est donc le `i`+1ième). L'intérêt de ceci est que l'on peut accéder à un élément variable, dont l'indice est lui-même calculé. Voici par exemple un calcul de produit scalaire de vecteurs à trois composantes :

```
float produit_scalaire (float a[3], float b[3])
{
  float resultat;
  int i;

  resultat = 0.0;
  for (i=0; i<3; i++)
    resultat = resultat + a[i]*b[i];
  return resultat;
}
```

(La boucle `for (i=0; i<3; i++)` itère l'instruction qui suit pour `i` allant de 0 inclus à 3 exclu.)

Pour changer le contenu des éléments d'un tableau, l'instruction d'affectation est en fait plus générale que celle que nous avons vue plus haut. On peut donc notamment écrire

```
a[0] = 5.0;
```

pour mettre le flottant `5.0` en premier élément du tableau (sans toucher aux autres). On peut ainsi effectuer une multiplication matricielle comme suit, par exemple :

```
  float a[3][3]; /* Multiplication de matrices 3x3, a et b,
                    résultat dans c. */
  float b[3][3]; /* A noter que les tableaux bidimensionnels
                    sont juste des tableaux à trois éléments
                    de tableaux à trois éléments. */
  float c[3][3];
```

```
  int i,j,k;

  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      {
        c[i][j] = 0.0;
        for (k=0; k<3; k++)
          c[i][j] = c[i][j] + a[i][k]*b[k][j];
      }
```

Les autres structures de données importantes en C sont les structures, les unions, et les pointeurs. Je ne parlerai pas des pointeurs ici, et je préfère attendre que vous ayez vu un peu d'architecture des machines au travers du langage machine en leçon 3 d'abord ; les pointeurs ne se comprennent vraiment qu'une fois qu'on sait comment la machine fonctionne vraiment, à un niveau plus bas.

Les *structures*, appelées aussi *enregistrements* ("records") sont essentiellement des n-uplets. Par exemple, on peut définir le type des nombres complexes en C comme

```
struct complex {
  float re;
  float im;
};
```

Ceci définit un nombre complexe comme étant un couple de deux flottants (sa partie réelle et sa partie imaginaire). Étant donnée une variable complexe `z`, typiquement déclarée par :

```
struct complex z;
```

on pourra accéder à sa partie réelle en écrivant `x.re`, et à sa partie imaginaire par `x.im`. Les identificateurs `re` et `im` sont appelés les *champs* de la structure `z`. De même que pour les tableaux, on pourra affecter (la valeur d')une expression à chaque champ individuellement. Par exemple,

```
struct complex x, y, z;

z.re = x.re + y.re;
z.im = x.im + y.im;
```

calcule dans `z` la somme des deux nombres complexes `x` et `y`.

▸ **EXERCICE 2.6**
Écrire de même la différence, le produit, et la division de deux nombres complexes.

▸ **EXERCICE 2.7**
Écrire un programme de multiplication de deux matrices 3×3 de nombres complexes (à compléter) :

```
struct complex a[3][3];
struct complex b[3][3];
struct complex c[3][3];
int i, j, k;

for (i=0; ...
```

Les structures C permettent de décrire des *produits cartésiens* de types. Si $A_1, \ldots, A_n$ sont des types C, le type `struct toto {` A_1 `a1;` $\ldots A_n$ `an; }` est essentiellement un type de n-uplets d'objets pris dans $A_1, \ldots, A_n$.

On peut aussi décrire quelque chose qui ressemble à des unions de deux types :

```
union nombre {
  int i;
  float x;
};
```

par exemple décrit le type `union nombre` comme consistant en des objets qui sont des entiers machine ou bien des flottants. Je vous déconseille d'utiliser cette construction avant de bien comprendre ce qu'elle fait vraiment : consultez n'importe quel livre d'introduction à C. Attention, cette construction n'a en fait pas grand-chose à voir avec l'union ensembliste, malgré les apparences. Elle permet d'autre part toute une série de hacks (et toute une série de bogues aussi) que je ne peux décemment pas vous recommander.

2.1.3 Structures de contrôle

Pour organiser les affectations en séquence, le langage C, comme la plupart des autres langages impératifs, dispose d'un certain nombre de constructions de base, étendues par des formes dérivées (du *sucre syntaxique*) :

- l'instruction qui ne fait rien (!) : elle se note `;`, ou bien `{}` ;
- la *composition séquentielle*, ou *séquence* : c_1`;`c_2`;` exécute l'instruction c_1, puis e_2. Par exemple, `x=3; y=x+1;` commence par ranger 3 dans `x` ; une fois ceci fait, ceci calcule `x+1`, soit 4, et le range dans `y` ;
- le *test*, aussi appelé la *conditionnelle* :

  `if (`e`)` c_1 `else` c_2

  commence par calculer la valeur de l'expression `e`, qui est censée être un entier machine. Cet entier représente une condition booléenne (vrai/faux). Si cet entier est non nul (ce qui par convention signifie que la condition est vraie), alors c_1 est executée, sinon c_2 est exécutée. Par exemple,

  ```
  if (x==3)
    printf ("x vaut bien 3.\n");
  else printf("Ah, x ne vaut pas 3, tiens.\n");
  ```

À noter ici que la sémantique de `x==3` n'est pas exactement de retourner vrai si `x` contient l'entier 3, et faux sinon ; en fait, `x==3` retourne l'entier 1 si `x` vaut 3, et 0 sinon. D'autres langages impératifs ont un type spécial (`bool` en Pascal, par exemple) pour dénoter les valeurs de vérité, C non : en C, les booléens sont codés sous forme d'entiers.

– La boucle *while* :

$$\texttt{while (}e\texttt{)}\ c$$

commence par calculer e. Si e est vrai (i.e., un entier non nul), alors c est exécutée. À la différence de la conditionnelle `if`, une fois l'exécutioin de c terminée, la boucle revient au début, recalcule e : si e est vrai de nouveau, c est reexécuté, et ainsi de suite, jusqu'à temps que e devienne faux (le cas échéant).

On a déjà vu en leçon 1 que la boucle était sémantiquement une conditionnelle plus un point fixe :

$$\texttt{while (}e\texttt{)}\ c \ = \ \texttt{if (}e\texttt{)}\ \{c; (\texttt{while (}e\texttt{)}\ c)\ \}$$

Et c'est tout ! On peut calculer tout ce qui est calculable avec ces seules instructions.

C propose aussi quelques extensions syntaxiques. Notamment, le *bloc*

$$\begin{array}{l} \{ \\ \quad c_1; \\ \quad \ldots \\ \quad c_n; \\ \} \end{array}$$

permet de voir n'importe quelle composition d'instructions $c_1, \ldots, c_n$ comme si ce n'était qu'une (grosse) instruction. Ceci permet notamment d'écrire :

```
if (x==3)
  {
    y = z;
    if (z==4)
      printf ("Non seulement x vaut 3 mais z vaut 4.\n");
    else printf ("z ne vaut pas 4, mais y vaut z maintenant.\n");
  }
else
  printf ("x ne vaut pas 3.\n");
```

On peut aussi éviter d'écrire la partie `else` d'un test dans certains cas : `if (`e`)` c est une abréviation de `if (`e`)` c `else ;`.

Une autre construction très fréquente est la boucle *for* :

$$\texttt{for (}e_1; e_2; e_3\texttt{)}\ c$$

où e_1, e_2, e_3 sont des expressions (optionnelles : on peut ne rien écrire à la place de chacune), et c une instruction. Ceci est *exactement* équivalent à

$$e_1; \texttt{while (}e_2\texttt{)}\ \{\ c; e_3;\ \}$$

▸ **EXERCICE 2.8**
Réécrire la fonction `fact` du début de ce cours en terme de boucle *while*, comme indiqué ci-dessus. Ceci est-il lisible ?

▸ **EXERCICE 2.9**
L'idée de la construction `for` de C est de simuler une boucle pour `i` variant de m à n, comme la construction `for i=`a `upto` b `do`... du langage Pascal ou la construction `for i=`a `to` b `do` ... `done` de OCaml, en écrivant à la place `for (i=`a`; i<=`b`; i++)` .... Cependant, en C, il est légal d'écrire :

```
for (i=1; i<=5; i++)
{
  printf ("La valeur de i est %d.\n", i);
  i = i+1;
}
```

Notez que la bizarrerie est que l'on change la valeur de `i` dans le corps de la boucle une seconde fois, en avant-dernière ligne. Que fait ce bout de code ? Peut-on faire pareil en OCaml ?

Finalement, en C on a aussi une construction `switch`. Disons en première approche que le code :

```
switch (e) {
       case ⟨constante 1⟩: c1; break;
       case ⟨constante 2⟩: c2; break;
       ...
       case ⟨constante n⟩: cn; break;
       default: c; break;
```

fait la même chose que (où `i` est une variable fraîche) :

```
{
 int i;
 i = e;
 if (i==⟨constante 1⟩)
     c1 ;
 else if (i==⟨constante 2⟩)
          c2 ;
 else if ...
     ...
 else if (i==⟨constante n⟩)
          cn ;
 else c;
```

La construction `switch` est censée être plus rapide que la suite de `if` ci-dessus. Elle admet aussi quelques variantes, notamment, si l'on omet le mot-clé `break` en fin d'une ligne `case`⟨constante i⟩, l'exécution de c_i se continuera sur celle de c_{i+1}. C'est une source courante d'erreurs en C, et un des charmes particuliers du langage.

2.2 Langages fonctionnels, le cas de mini-Caml

Il existe aussi en C une notion de procédure que l'on peut appeler pour faire un calcul auxiliaire, et qui une fois le calcul fait retourne un résultat. C'est ce que nous avons déjà fait dans le programme `fact`.

Les langages *fonctionnels*, et notamment Caml et ses variantes (vous connaissez en général tous Objective Caml, aussi appelé OCaml) sont centrés sur cette notion de fonction. Il y a de nombreuses bonnes références sur OCaml, que l'on peut notamment trouver sur la page `http://cristal.inria.fr/`.

Nous allons plus particulièrement étudier une version réduite d'OCaml, que nous nommerons mini-Caml dans la suite. Il s'agit au passage du langage dont vous aurez à écrire un compilateur, puis un système d'inférence de types, en projet. Le typage vous sera enseigné par Delia Kesner. Dans la suite de ces notes, je considérerai une version non typée.

L'essentiel d'un langage fonctionnel est que l'on peut y écrire des fonctions : `fun` x `->` M, où M est une expression du langage, est encore une expression du langage, et dénote la fonction qui à x associe M. On appellera une telle construction syntaxique `fun` x `->` M une *abstraction*. Ceci est (pratiquement) la notion mathématique de fonction, comme on le verra quand on en donnera la sémantique un peu plus loin. Par exemple, `fun x -> x+1` est la fonction qui à tout entier n associe l'entier $n + 1$.

On a d'autre part une construction, dite d'*application* : si M dénote une fonction et N dénote un argument possible de la fonction, alors MN dénote le résultat de l'application de la fonction M à l'argument N. En particulier, `(fun x -> x + 1) (3)`, que l'on peut aussi noter `(fun x -> x + 1) 3`, a bien pour valeur `4`. (On peut rajouter des parenthèses à volonté autour de sous-expressions pour éliminer les ambiguïtés syntaxiques, comme en C d'ailleurs.)

Le sous-langage de mini-Caml où l'on ne peut écrire que des variables, des abstractions et des applications s'appelle le *λ-calcul*, et a été inventé et étudié par Alonzo Church et ses successeurs à partir des années 1930. C'est aujourd'hui un outil fondamental en logique et en informatique théorique ; c'est au passage (pub !) le sujet du cours de logique et informatique du second semestre, en commun avec l'ENS de la rue d'Ulm.

En OCaml, on a en plus une construction permettant de donner des noms intermédiaires à des variables : `let` x=M `in` N a pour effet de calculer la valeur de M, de lui donner le nom x, puis de calculer N. Dans N, on a le droit de se référer à x pour dénoter la valeur de M. Au passage, ceci pourrait s'écrire autrement, sous forme de l'expression `(fun` x `->` N`)` M, ce serait équivalent. . . à part que le système de typage de Caml, que vous verrez avec Delia Kesner, considère la forme `let` d'une façon un peu spéciale, et différente de la forme utilisant `fun` et l'application.

On retrouve, comme dans les expressions C, toute une famille de constantes et d'opérations permettant de faire des calculs : les constantes entières `0`, `1`, . . ., `-1`, `-2`, . . ., les sommes $M+N$, les différences $M-N$, les produits $M*N$, les quotients entiers (ou divisions flottantes) M/N, les restes M `mod` N, . . ., avec essentiellement la même sémantique qu'en C. (Nous considérerons que ces opérations ont réellement la même sémantique qu'en C. En OCaml, pour des raisons techniques, les entiers ne vont que de -2^{30} à $2^{30} - 1$ sur une architecture 32 bits, et l'on calcule modulo 2^{31}, pas 2^{32}.)

Une différence importante avec C est que *la valeur des variables ne change pas*. En C, on peut toujours changer la valeur d'une variable, disons `x`, en écrivant par exemple `x=1`. Il n'y a aucune instruction permettant de changer la valeur d'une variable en OCaml ou en mini-Caml. Une conséquence est que, une fois calculée la valeur d'une variable, on peut être sûre qu'elle vaudra toujours la même valeur. Ceci est important pour la lisibilité des programmes. Dans un programme Caml de la forme `let x=3 in` N par exemple, on peut être sûr que toute référence à `x` dans N, aussi loin qu'on se trouve dans le source de la déclaration `let x=...`, vaudra `3`. En C, on peut écrire des instructions apparemment similaires :

```
{
  int x;

  x = 3;
  ... N ...
}
```

Mais pour être sûr que toutes les occurrences de `x` dans `N` se réfèrent bien à la valeur calculée `3`, il faudra d'abord s'assurer qu'aucune autre instruction n'a préalablement modifié la valeur de `x`. Ceci peut se révéler très compliqué.

Si OCaml et mini-Caml privilégient donc un style d'écriture mathématique, où les variables ne sont pas des cases où l'on range des valeurs comme en C, mais des noms dénotant des valeurs calculées par ailleurs, les langages de la famille Caml permettent cependant si on le souhaite d'écrire des affectations, plus ou moins comme en C. Pour ceci, on utilise la notion de *références*. En mini-Caml, l'expression `ref` M a pour effet d'*allouer* (de créer) une nouvelle référence, c'est-à-dire une nouvelle case mémoire. Cette case contient initialement la valeur de M. On peut ensuite relire le contenu d'une case mémoire (dénotée par N, disons) en écrivant l'expression `!`N. On peut aussi effectuer une affectation, et N`:=`P a pour effet de calculer la valeur de N, qui doit être une référence r, de calculer P, et de stocker la valeur de P à l'intérieur de la référence r.

Par exemple,

```
let x = ref 0
in !x
```

créer une référence contenant l'entier `0`, que l'on repère par le nom `x`, puis relit le contenu de cette référence : le résultat est `0`. Essayons une affectation :

```
let x = ref 0
in (x := 3; !x)
```

Ceci crée une référence contenant `0`, remplace son contenu par `3`, puis lit ce contenu : le résultat est `3`. Un peu plus compliqué :

```
let x = ref 3
in (x := !x+1; !x)
```

Ceci crée une référence contenant `3` ; l'effet de `x := !x+1` est d'ajouter un au contenu de cette référence, et le résultat final `!x` est donc `4`.

On a ici utilisé la construction `;` qui, comme en C, définit une composition séquentiel d'instructions (ou, comme ici, d'expressions). On dispose aussi d'une conditionnelle `if` M `then` N `else` P, qui calcule N si M est vrai et P sinon. (On considérera comme en C que seul l'entier 0 dénote le faux.) La valeur de la conditionnelle est celle de N dans le premier cas, celle de P dans le second. On notera que les conditionnelles ont une valeur en Caml, alors qu'elles étaient des instructions, qui n'ont pas de valeur, en C. (Pour dire la vérité, C dispose d'une autre instruction conditionnelle pour les expressions, dont la syntaxe est $e_1 ? e_2 : e_3$.)

On n'inclura pas de construction de boucle *while* en mini-Caml. L'exemple de la factorielle écrite en C et en Caml de la section 1.1 vous a peut-être suggéré que l'on pouvait coder la boucle *while* de la factorielle C en une construction `let rec` (définition de fonction récursive) de Caml. C'est effectivement le cas. Une construction `while (`e`)` c de C est grosso modo équivalente à la construction mini-Caml

```
letrec boucle = fun () ->
       if e
          then (c; boucle ())
       else ();;
boucle ()
```

qui définit d'abord un point fixe du `if` sous forme d'une fonction `boucle` à un argument (bidon), puis demande la valeur `boucle()` du point fixe.

La construction `letrec` f=M`;;` sera la seule construction de programme mini-Caml (les constructions vues plus haut sont des constructions d'expressions). Elle définit f comme valant exactement la même chose que M, et ce même si M fait appel à f. Pour des raisons techniques, on restreindra M à être elle-même la définition d'une fonction `fun` $x \to N$. Une définition via `letrec` (par exemple, pour `boucle`) de f est une *définition récursive*, c'est-à-dire que f est définie en fonction d'elle-même. Ceci signifie bien sûr que f doit être définie comme un point fixe de la fonction `fun` f `->` M.

La construction `letrec` de mini-Caml correspond au `let rec` d'OCaml. La construction `let` x=M`;;` en OCaml, qui définit x comme valant M *non récursivement* (M ne peut pas utiliser la valeur de x que l'on est en train de définir) correspond à `let` x=M`;;` en mini-Caml.

On termine cette description de l'essentiel d'OCaml et de mini-Caml en mentionnant que ces langages disposent d'une construction de produit cartésien : l'expression $(M_1, \ldots, M_n)$ dénote le n-uplet formé des valeurs respectives des expressions M_1, ..., M_n dans cet ordre. D'autre part, l'expression $\mathtt{proj}_1 M$ permettra d'extraire la première composante du n-uplet M (si c'en est un et si $n \geq 1$), $\mathtt{proj}_2 M$ retrouvera la deuxième composante (si $n \geq 2$), et ainsi de suite. Il s'agit d'un vrai produit cartésien. En particulier, contrairement aux `struct` de C, il est impossible de changer après coup la valeur d'une des composantes d'un n-uplet.

On résume la syntaxe de mini-Caml en figure 4.

En guise d'exemple résumant notre discussion informelle de la sémantique de mini-Caml ci-dessus, voici une réalisation possible de `cat` en mini-Caml :

			Termes (fonctions, données)
$M, N, P, ...$	$::=$	x	variables (*non* modifiables)
	$\mid$	MN	application de M à N
	$\mid$	$\texttt{fun}\, x \to M$	fonction qui à x associe M
	$\mid$	$\texttt{let}\, x = M\, \texttt{in}\, N$	définition
	$\mid$	$\texttt{ref}\, M \mid !M \mid M{:=}N$	références (modifiables)
	$\mid$	$\texttt{0} \mid \texttt{1} \mid \ldots \mid M{+}N \mid \ldots$	arithmétique
	$\mid$	$(M_1, ..., M_n) \mid \texttt{proj}_i M$	n-uplets
	$\mid$	$M; N$	séquence
	$\mid$	$\texttt{if}\, M\, \texttt{then}\, N\, \texttt{else}\, P$	tests
$Prog$	$::=$	ϵ	Programmes
	$\mid$	$Prog \quad \texttt{letrec}\, f = \texttt{fun}\, x \to M;;$	
	$\mid$	$Prog \quad \texttt{let}\, x = M;;$	

FIG. 4 – La syntaxe de mini-Caml

```
letrec boucle_interne = fun f ->
       let c = fgetc f
       in if c=EOF
             then 0
          else (putchar c; boucle_interne f);;
letrec boucle_externe = fun i -> fun argc -> fun argv ->
       if i<argc
          then let nom = sub argv i
            in let f = fopen nom "r"
            in (boucle_interne f; fclose f;
                boucle_externe (i+1) argc argv)
       else 0;;
letrec main = fun argc -> fun argv ->
       boucle_externe 1 argc argv;;
```

Ceci suppose que `fgetc` est, comme en C, une fonction prenant un fichier `f` en argument et retournant le prochain caractère lu dans ce fichier. De même, on suppose que `EOF` a été prédéfini comme la constante dénotant la fin de fichier (autrement dit, -1). On suppose aussi que `putchar` prend un caractère `c` et l'affiche, que `fopen` ouvre un fichier dont le nom est en premier argument et le mode d'ouverture (ici, `"r"`) est en second, que `fclose` ferme un fichier donné en argument. Autrement dit, on supposera qu'on a accès en mini-Caml à toutes les fonctions standard disponibles par ailleurs en C. (Ce sera le cas dans votre projet.) Finalement, on suppose que `sub argv i` récupère l'élément numéro `i` du tableau `argv`.

On pourra remarquer dans cet exemple le codage des fonctions à plusieurs arguments. On a choisi, comme il est traditionnel en Caml, de coder les fonctions à deux arguments comme des fonctions prenant leur premier argument, et retournant une nouvelle fonction qui prend le deuxième argument et effectue le calcul. Par exemple, le point d'entrée `main` du programme est défini comme une fonction prenant un paramètre `argc` en argument, retournant une fonction prenant un paramètre `argv` en argument, puis appelant `boucle_externe`.

On aurait pu aussi définir les fonctions binaires comme des fonctions prenant des couples (M_1, M_2) en argument. Ce serait facile en OCaml, grâce à l'utilisation de *filtres d'entrée* ("patterns"), mais déjà moins commode en mini-Caml, qui se veut un langage relativement minimal.