Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

P.ex.,

```
# let mot_4 = concatener mot_3 mot_3 in
string_of_mot mot_4 (fun x -> x);;
- : string = "abbabb"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))


# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))


# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
# let w = mot_fibo 5 in string_of_mot w (fun x -> x);;
- : string = "abaababa"
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y then
            {rouge = 127; vert = 127;
                bleu = 127}
        else
            {rouge = 255; vert = 255;
                bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

Exemple complet 2 : images fonctionnelles

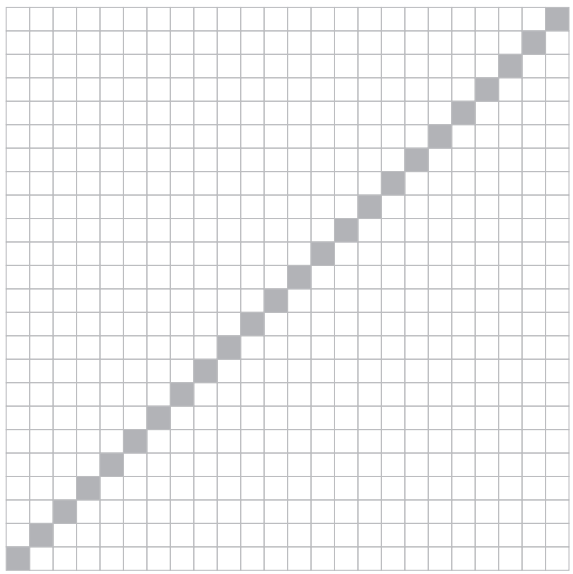
Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y then
            {rouge = 127; vert = 127;
                bleu = 127}
        else
            {rouge = 255; vert = 255;
                bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

représente l'image



Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Voici une fonction qui calcule l'image miroir d'une image en entrée :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x + 1, y)
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Voici une fonction qui calcule l'image miroir d'une image en entrée :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x + 1, y)
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Toutes ces fonctions s'évaluent en temps $\Theta(1)$. Cette complexité ne dépend donc pas de la taille de l'image !

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers.

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Question : comment représenter des séries génératrices ?

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Par exemple, la série génératrice des puissances de 2 est ainsi codée par

```
let puissances_2 =
    (fun n -> (int_of_float (2. ** (float_of_int n))));;
```

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$

3. **produit** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \cdot \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \sum_{i=0}^{k} \alpha_i \beta_{k-i} t^k .$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$

3. **produit** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \cdot \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \sum_{i=0}^{k} \alpha_i \beta_{k-i} t^k .$$

Il est possible de les implanter simplement en utilisant les fonctions d'ordre supérieur.

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;

# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;

# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
# (s3 3);;
- : int = 16
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;

# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
# (s3 3);;
- : int = 16
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =
    (fun k -> (s1 k) * (s2 k));;
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>

# (sg_un_carre 0), (sg_un_carre 1), (sg_un_carre 2), (sg_un_carre 3);;
- : int * int * int * int = (1, 2, 3, 4)
```

Plan

Notions

- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

- Une **valeur polymorphe** est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé. P.ex.,

```
# Vide;;
- : 'a liste = Vide
```

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de `x`), alors `x` est du type le plus général `'a`.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
`snd : 'a * 'b -> 'b`	Renvoie la 2e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
    if n = 0 then
        1
    else
        let tmp = (puiss x (n / 2)) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x;;
```

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
    if n = 0 then
        1
    else
        let tmp = (puiss x (n / 2)) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x;;
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;
- Le type de retour est `'e`.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;
- Le type de retour est `'e`.

La valeur renvoyée est $\texttt{x}^{\texttt{n}}$.

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
# (puiss_poly (fun u v -> u ^ v) "" "abb" 4);;
- : string = "abbabbabbabb"
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
# (puiss_poly (fun u v -> u ^ v) "" "abb" 4);;
- : string = "abbabbabbabb"
# (puiss_poly (fun a b -> a || b) false false 293898273);;
- : bool = false
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2e paramètre dans le 1er.

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2e paramètre dans le 1er.

```
let rec appartient_liste lst x =
    match lst with
        |Vide -> false
        |(Cellule (y, _)) when y = x -> true
        |(Cellule (_, reste)) -> (appartient_liste reste x);;
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

`'e liste -> 'e -> bool`

testant la présence de la valeur du 2e paramètre dans le 1er.

```
let rec appartient_liste lst x =
    match lst with
        |Vide -> false
        |(Cellule (y, _)) when y = x -> true
        |(Cellule (_, reste)) -> (appartient_liste reste x);;
```

Ceci se base sur le fait que test d'égalité `=` est polymorphe (de type `'a -> 'a -> 'a` ; il n'est donc pas à fournir en paramètre à la fonction).

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

`('e -> 'e -> 'e) -> 'e liste -> 'e`

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

```
let maximum_liste mmax lst =
    let rec aux lst max_prefixe =
        match lst with
            |Vide -> max_prefixe
            |(Cellule (x, reste)) -> (aux reste (mmax max_prefixe x))
    in
    match lst with
        |Vide -> (failwith "liste vide")
        |(Cellule (x, reste)) -> (aux reste x);;
```

Plan

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison ;

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison ;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution termine dans ce cas.

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (f 1 (f 2 1 4) 12)
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (f 1 (f 2 1 4) 12)
          ⇝ (f 1 6 12)
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (f 1 (f 2 1 4) 12)
          ⇝ (f 1 6 12)
          ⇝ 13
```

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;

2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;

2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont le Caml.

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (1 * 1) + (3 * 4)
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (1 * 1) + (3 * 4)
          ⇝ 13
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
          ⇝ (4 * 3) * (4 * 3) + (2 * 1)
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
            ⇝ (4 * 3) * (4 * 3) + (2 * 1)
            ⇝ 146
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
            ⇝ (4 * 3) * (4 * 3) + (2 * 1)
            ⇝ 146
```

L'argument `(4 * 3)` est évalué ainsi deux fois (au lieu d'une seule que ferait un appel par valeur).

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un `ai`, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de `f` correspondant à un `ai` ne redemande pas d'être réévaluée.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage Haskell utilise cette stratégie.

Plan

Plan

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

`[e1 ; e2 ; ... ; en]`

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;
- : int list = [2; 4; 8; 16]
```

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

`[e1 ; e2 ; ... ; en]`

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;
- : int list = [2; 4; 8; 16]
```

La **liste vide** est notée `[]`.

```
# [];;
- : 'a list = []
```

Les listes

Le langage Caml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

`[e1 ; e2 ; ... ; en]`

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;
- : int list = [2; 4; 8; 16]
```

La **liste vide** est notée `[]`.

```
# [];;
- : 'a list = []
```

C'est une valeur polymorphe.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

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

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

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

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

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

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

P.ex.,

```
# 2 :: [1 ; 2 ; 3];;
- : int list = [2; 1; 2; 3]
```

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

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

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

P.ex.,

```
# 2 :: [1 ; 2 ; 3];;
- : int list = [2; 1; 2; 3]
```

Il est **associatif de droite à gauche**.

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

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

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

P.ex.,

```
# 2 :: [1; 2; 3];;
- : int list = [2; 1; 2; 3]
```

Il est **associatif de droite à gauche**.

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

L'expression `1 :: 2 :: 3 :: []` désigne en effet l'expression totalement parenthésée `(1 :: (2 :: (3 :: [])))`.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =
      match lst with
          |[] -> (failwith "liste vide")
          |e :: _ -> e;;
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1er élément.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =
      match lst with
          |[] -> (failwith "liste vide")
          |e :: _ -> e;;
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1er élément.

```
# let rec un_sur_deux lst =
      match lst with
          |[] -> []
          |[e] -> [e]
          |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;
val un_sur_deux : 'a list -> 'a list = <fun>
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =
      match lst with
          |[] -> (failwith "liste vide")
          |e :: _ -> e;;
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1er élément.

```
# let rec un_sur_deux lst =
      match lst with
          |[] -> []
          |[e] -> [e]
          |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;
val un_sur_deux : 'a list -> 'a list = <fun>
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : ce sont des fonctions polymorphes.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes.

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;
4. **combiner** les éléments d'une liste pour calculer une valeur ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;
4. **combiner** les éléments d'une liste pour calculer une valeur ;
5. **permuter** les éléments d'une liste.

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;

4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;

4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;

5. **[permutation]** tri d'une liste, image miroir d'une liste.

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;
2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;
3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;
4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;
5. **[permutation]** tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de fonctions d'ordre supérieur.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]

# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;
- : int list = [97; 98; 99; 49; 50; 51]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

`('a -> 'b) -> 'a list -> 'b list`

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]

# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;
- : int list = [97; 98; 99; 49; 50; 51]
```

Nous avons réimplanté ici la fonction `map` du module `List`.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

`('a -> bool) -> 'a list -> 'a list`

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;
- : int list = [8; 8; 6; 2]
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

`('a -> bool) -> 'a list -> 'a list`

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `filter` du module `List`.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

`('a -> bool) -> 'a list -> bool`

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a"; "aba"; "abacaba"; "abacabadabacaba"];;
- : bool = true
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a"; "aba"; "abacaba"; "abacabadabacaba"];;
- : bool = true
```

Nous avons réimplanté ici la fonction `for_all` du module `List`.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

`('a -> bool) -> 'a list -> bool`

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

`('a -> bool) -> 'a list -> bool`

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;
- : bool = false
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

`('a -> bool) -> 'a list -> bool`

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;
- : bool = false
```

Nous avons réimplanté ici la fonction `exists` du module `List`.

Association des éléments d'une liste

Le problème est le suivant.

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et que $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et que $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

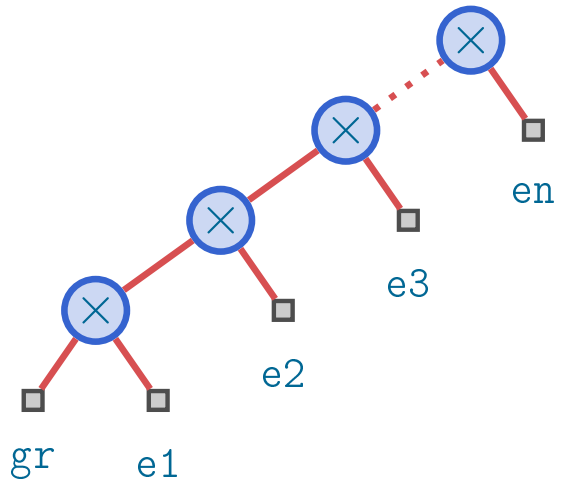
$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et que $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;
- lorsque `gr = e1`, `lst` est une liste dont les éléments sont comparables et $\times$ est la fonction qui renvoie **le plus grand** de ses deux arguments, ceci calcule la plus grande valeur de `lst`.

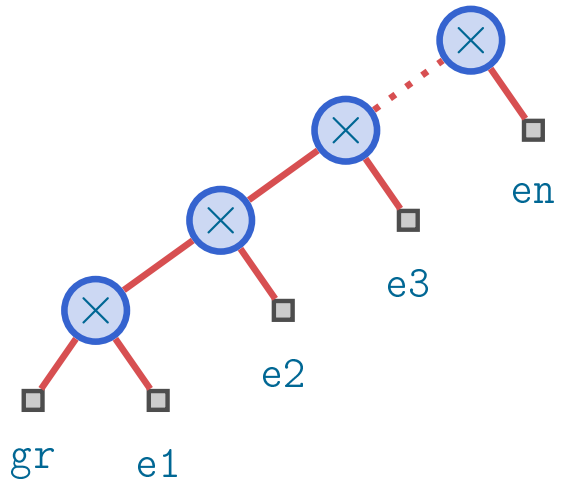
Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Pliage à gauche

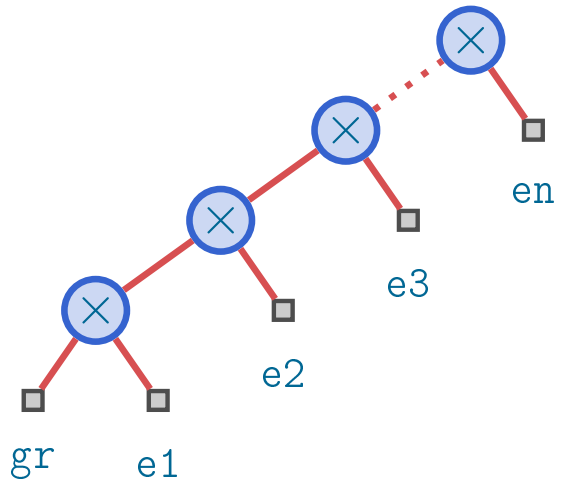
En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



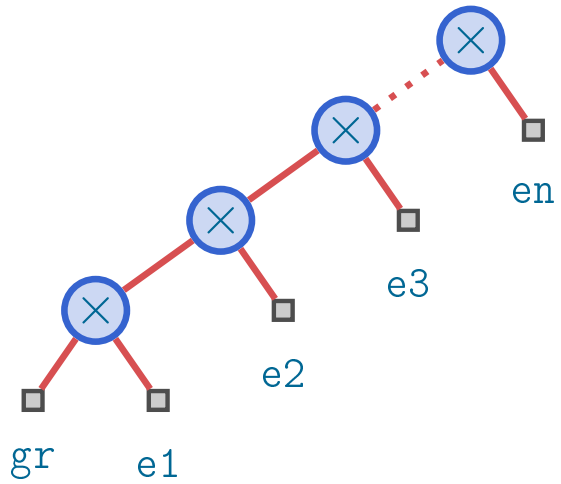
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



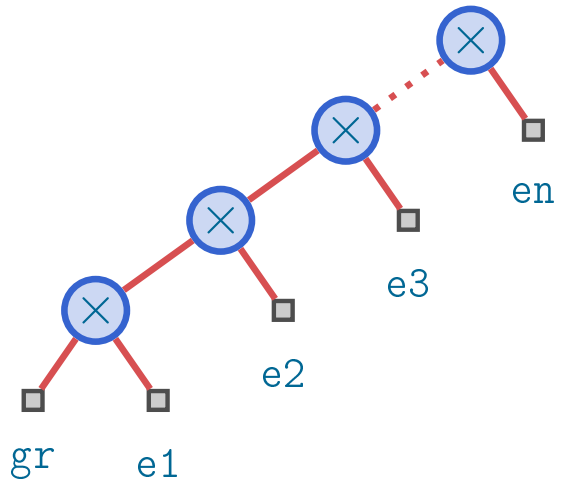
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;
2. la graine `gr` ;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;
2. la graine `gr` ;
3. la liste `lst = [e1 ; e2 ; e3 ; ... ; en]` des opérandes.

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11

# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11

# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5

# (pliage_gauche (^) "" ["mi" ; "la" ; "re" ; "sol" ; "do" ; "fa"]);;
- : string = "milaresoldofa"
```