RAPPORT DE PROJET

PROJET INPF12

Étudiant : Arnaud GRASSIAN
Formation : ENSIIE
Date : Avril 2025

École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

Table des matières

Introduction

Ce rapport présente le développement d'un éditeur de texte en OCaml. Ce projet a pour objectif de manipuler des structures comme les zippers pour que l'éditeur soit capable d'interpréter et d'appliquer des actions clavier basiques : déplacement du curseur, insertion de caractères, suppression, retour à la ligne... L'interface graphique utilise le module `Graphics` (cf manuel d'utilisation).

```
    On définit le type ('a,'b) zipper :

    type ('a,'b) zipper =
  { before:  'a list (* liste des éléments précédents dans l'ordre inversé *)
    ; current: 'b
    ; after:   'a list (* liste des éléments précédents dans l'ordre *)
    ; pos:     int (* position courrante *)
    }
;;
```

Spécifications

L'objectif du projet était de créer un éditeur de texte avec des fonctions simples : insérer ou effacer un caractère, se déplacer dans le texte. La notion principale à manipuler est la notion de buffer. Un buffer est un ensemble de lignes. Un ligne est un ensemble de caractères. Ainsi on définit un buffer comme un zipper avec une ligne courante, les lignes précédentes et les lignes suivantes. Une ligne est également définit comme un zipper avec un curseur, les caractères qui le précèdent et les caractères qui le suivent.

Par exemple, on définit la ligne vide :

```
let empty_line = { before = []; current = Cursor; after = []; pos = 0 };;
```

et le buffer vide :

```
let empty_buf  = { before = []; current = empty_line; after = []; pos = 0 };;
```

avec :

```
type line   = (char, cursor) zipper;;
type buffer = (line, line) zipper;;
```

Chaque action (déplacement du curseur, suppression, retour à la ligne...) se traduit par une transformation de l'état du buffer textuel, modélisé par cette double structure de zipper.

Implémentation

Le code est divisé en plusieurs parties :

2.1. Fonctions de base sur les zippers

Ces fonctions permettent de manipuler un zipper :

- `get_current,get_pos` : élément courant et position du zipper ;
- `fold_zipper` : fold sur les zipper :

  Par exemple, si on prend un `zipper` avec les éléments suivants :
  - `before` = ['b' ; 'a']
  - `current` = 'c'
  - `after` = ['d' ; 'e']

  Nous appelons `fold_zipper` avec :
  - `f` : une fonction qui ajoute l'élément en tête d'une liste (`fun c acc -> c :: acc`)
  - `g` : une fonction similaire appliquée à l'élément courant
  - `acc0` : la liste vide `[]`

  1. **Traitement de `before` avec `fold_right` :**
     - Remise dans l'ordre logique : 'a', puis 'b'
     - Accumulateur après cette étape : `['a'; 'b']`
  2. **Traitement de `current` avec `g` :**
     - Ajout de 'c' en tête
     - Accumulateur après cette étape : `['c'; 'a'; 'b']`
  3. **Traitement de `after` avec `fold_left` :**
     - Ajout successif de 'd', puis 'e'
     - Résultat final : `['c'; 'a'; 'b'; 'd'; 'e']`

- `update_with` : transformation de l'élément courant ;
- `move_left_line`, `move_right_line`, `move_left_buffer`, `move_right_buffer` : décalage du curseur.

2.2. Mouvements dans le buffer

Les fonctions `move_left1`, `move_right1`, `move_up`, `move_down` permettent de déplacer le curseur dans l'espace du texte.

La difficulté principale ici a été de conserver une position horizontale constante lors des montées/descentes dans les lignes, même si celles-ci sont de longueurs inégales. Cela a été résolu via une fonction auxiliaire `take` qui reconstitue les parties gauche et droite de la ligne cible (du haut ou du bas) en fonction de la position souhaitée.

Considérons un `buffer` contenant les lignes suivantes :

- `before` = [`ligne2` ; `ligne1`] (la ligne la plus proche est `ligne2`)
- `current` = `ligne3`
- `after` = [`ligne4` ; `ligne5`]

Le curseur est placé à la position 5 dans `ligne3` (après les 5 premiers caractères).

```
let move_up    (buf : buffer) : buffer  =
let length = List.length buf.current.before in match buf.before with
| [] -> buf
| p::q ->
{    after=buf.current::buf.after;
    pos=buf.pos-1;
    before=q;
    current={
      before=List.rev(fst (take length ((List.rev p.before)@p.after)));
      current=Cursor;
      after=snd(take length ((List.rev p.before)@p.after));
      pos=min length (List.length p.before + List.length p.after);
    }}
  ;;
```

Vérif : Le `before` n'est pas vide, on peut donc se déplacer vers le haut.

Récupération de la ligne du haut : On extrait `ligne2` (p) de `before`.

Curseur : On essaie de placer le curseur à la même position horizontale qu'avant (position 5). Si `ligne2` a moins de 5 caractères, on positionne le curseur à la fin de la ligne. c'est pourquoi on choisit pour la position du curseur le min entre la longueur de ligne3 et

celle de ligne2chesch. On pense bien à l'ordre de before (en inversant les before). `current` devient `ligne2` ajustée avec le curseur repositionné. `ligne3` est déplacée dans `after`. `before` est mis à jour en retirant `ligne2`.

`move_up` déplace le curseur vers la ligne précédente tout en essayant de conserver la position horizontale. Cela permet une navigation verticale naturelle dans l'éditeur, même lorsque les lignes ont des longueurs différentes.

2.3. Actions textuelles

La fonction `insert_char` permet d'ajouter un caractère (passé en argument) à la position courante dans la ligne et avance la position de 1.

La fonction `do_supr` permet de supprimer le caractère situé juste après la position courante.

`do_backspace1` supprime le caractère situé juste avant la position courant et recule le curseur d'un caractère et `create_newline` coupe la ligne courante en deux lignes au niveau du curseur de la ligne et insère une nouvelle ligne (le curseur du buffer se trouve sur cette nouvelle ligne).

2.4. Implantation plus ergonomique

Les fonctions `move_to_start_of_line` et `move_to_end_of_line` permettent respectivement de positionner le curseur au début ou à la fin de la ligne courante. Elles sont utiles pour gérer les cas où le curseur atteint les extrémités d'une ligne et qu'un déplacement (vers le haut ou vers le bas) est demandé. Elles permettent aussi, plus généralement, de rendre les déplacements verticaux cohérents, en maintenant une position logique du curseur malgré les différences de longueur entre les lignes.

Ensuite, les fonctions `move_left` et `move_right` ont été étendues pour prendre en compte les transitions inter-lignes : lorsqu'on tente de se déplacer vers la gauche alors que le curseur est au tout début d'une ligne, on passe automatiquement à la fin de la ligne précédente. Inversement, lorsqu'on se déplace vers la droite à la fin d'une ligne, le curseur est déplacé au début de la ligne suivante. Cela améliore fortement la fluidité de la navigation dans le

texte.

Les fonctions de suppression inter-lignes, `do_backspace` et `do_suppr`, ont également été repensées pour fusionner automatiquement deux lignes lorsque l'action de suppression est effectuée au début ou à la fin d'une ligne. Cela permet une manipulation intuitive du texte, similaire aux comportements observés dans les éditeurs classiques. Exemple d'utilisation de `do_suppr` Deux cas se présentent :

- Si la liste `after` de la ligne courante n'est pas vide, on supprime simplement le premier caractère de cette liste (`do_supr` codé précédemment).
- Si la liste `after` est vide mais qu'il existe une ligne suivante (`next_line`), alors la ligne courante est fusionnée avec la ligne suivante.

Voici le code de la fonction :

```
let do_suppr buf =
  let line = get_current buf in
  match line.after, buf.after with
  | [], next_line :: nexts ->
      {
        before = buf.before;
        current = {
          before = line.before;
          current = Cursor;
          after = line.after @ (List.rev next_line.before) @ next_line.after;
          pos = line.pos
        };
        after = nexts;
        pos = buf.pos
      }
  | _ -> do_supr buf
;;
```

Supposons un `buffer` où :

- La `current line` est : `["a"; "b"; Cursor; "c"]` (curseur après 'b', `after` = ['c'])
- Le `next line` est : `["d"; "e"; "f"]`

Si le curseur est placé avant 'c' :

- `do_suppr` supprime simplement 'c'.
- La ligne devient : `["a"; "b"; Cursor]`

Si le curseur est placé en fin de ligne (plus de caractères après) :

- `do_suppr` fusionne la ligne courante avec la ligne suivante. La nouvelle ligne devient : `["a"; "b"; Cursor; "d"; "e"; "f"]`. Il faut dans ce cas bien penser à modifier la ligne courante, avec pour nouveau `after` la concaténation de l'after de la ligne courante ([]), du before inversé de la ligne d'après (List.rev next_line.before) et de l'after de la ligne suivante next_line.after.

La logique est la même pour `do_backspace` : si le curseur est en début de ligne, la fonction fusionne avec la ligne précédente (sinon, `do_backspace1` supprime simplement le dernier caractère de `before`).

Lors de la fusion avec la ligne précédente, on construit la nouvelle liste `before` ainsi :

```
before = line.before @ (List.rev prev_line.after) @ prev_line.before;
```

- Dans un `zipper`, la liste `before` est stockée **dans l'ordre inversé**.
- La liste `after` est stockée **dans l'ordre normal**.

Il faut donc faire attention à l'ordre des éléments :

- `prev_line.before` est inversé dans la structure du `zipper`.
- `line.before` est aussi inversé.

Ainsi, pour obtenir la liste dans le bon sens (de gauche à droite dans le texte), on doit faire :

```
List.rev prev_line.before @ prev_line.after @ List.rev line.before
```

Par exemple, si on a :

- `prev_line` :
  - `before` = ["d" ; "c" ; "b" ; "a"] (ce qui correspond en réalité à "abcd")
  - `after` = []
- `current_line` :
  - `before` = []

— `after` = ["e" ; "f" ; "g"]

Le curseur est placé au tout début de la `current_line`.

Après l'appel à `do_backspace`, pour construire la nouvelle ligne :

— On inverse `prev_line.before` pour obtenir ["a" ; "b" ; "c" ; "d"]
— On laisse `prev_line.after` (vide ici)
— On laisse `current_line.after` ["e" ; "f" ; "g"]

Ainsi, la nouvelle `current_line` est :

— `before` = ["d" ; "c" ; "b" ; "a"]@[]@[] les deux listes vides correspondent respectivement à prev_line.after et List.rev line.before.
— `current` = Cursor
— `after` = ["e" ; "f" ; "g"]

En affichage logique (après inversions des `before`), cela donne :

"a", "b", "c", "d", **Cursor**, "e", "f", "g"

Le curseur est donc bien positionné entre le "d" et le "e".

Choix de structures et modélisation

3.1. Le type zipper

Le cœur de l'éditeur repose sur l'utilisation du type générique `('a,'b) zipper`, qui permet de représenter une séquence avec un curseur :

- Une liste `before` représentant les éléments précédents dans l'ordre inversé ;
- Un élément `current` sous le curseur ;
- Une liste `after` représentant les éléments suivants ;
- Un champ `pos` indiquant la position du curseur.

On a donc :

- Une `line` est un zipper de caractères ;

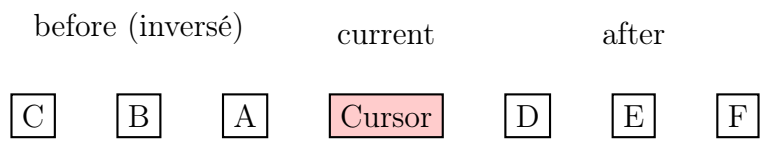

FIGURE 3.1 – Représentation d'une ligne (`line`) 'ABCDEF' avec le type `(char, cursor) zipper`

et

- Un `buffer` est un zipper de lignes.

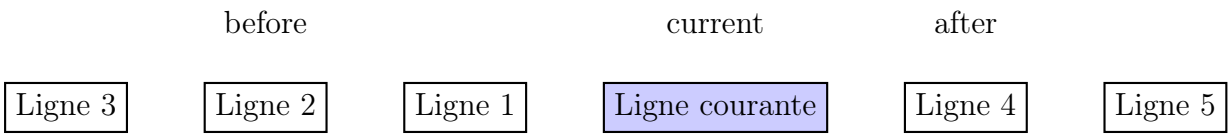

FIGURE 3.2 – Représentation d'un buffer (`(line, line) zipper`)

3.2. Avantages de cette structure

L'utilisation des zippers présente plusieurs intérêts :

— Accès constant aux éléments autour du curseur ;
— Facilité d'insertion ou de suppression sans coût de parcours.

Ce choix structurel a aussi permis une parfaite séparation entre les actions sur lignes et celles sur caractères.

Tests

Les tests ont été réalisés en exécutant l'éditeur avec `make` puis `./test` (voir manuel d'utilisation. Voici quelques scénarios de test :

4.1. Insertion de texte

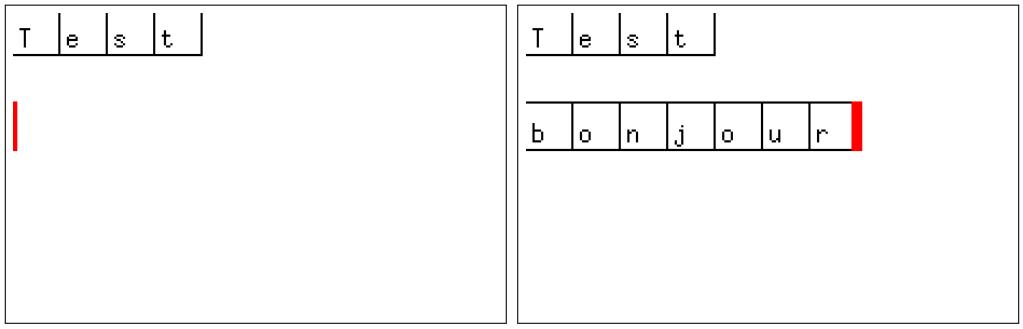

FIGURE 4.1 – Insertion du mot "bonjour"

4.2. Retour à la ligne

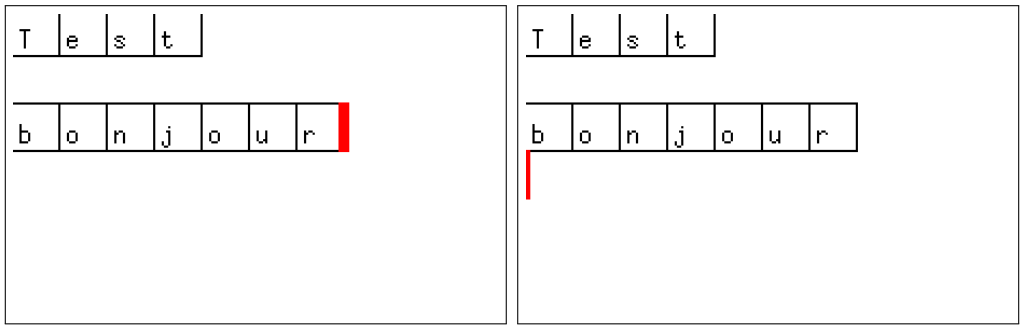

FIGURE 4.2 – Appui sur Entrée après "bonjour"

4.3. Déplacements verticaux

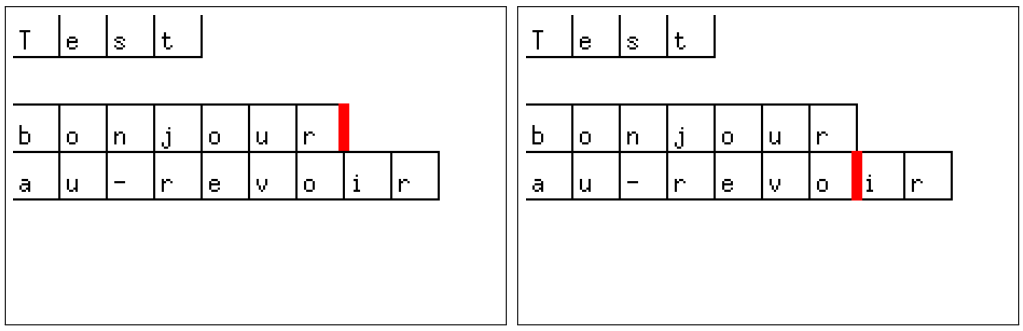

FIGURE 4.3 – CTRL-Z puis CTRL-S (haut puis bas)

4.4. Suppression

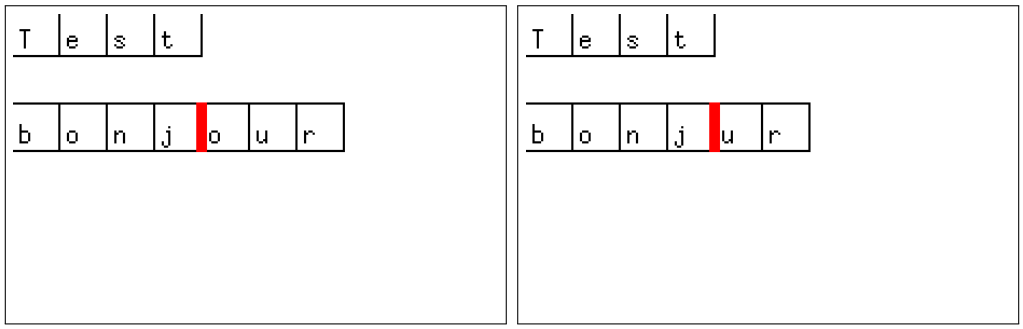

FIGURE 4.4 – Suppression d'un caractère avec la touche Suppr

4.5. Backspace

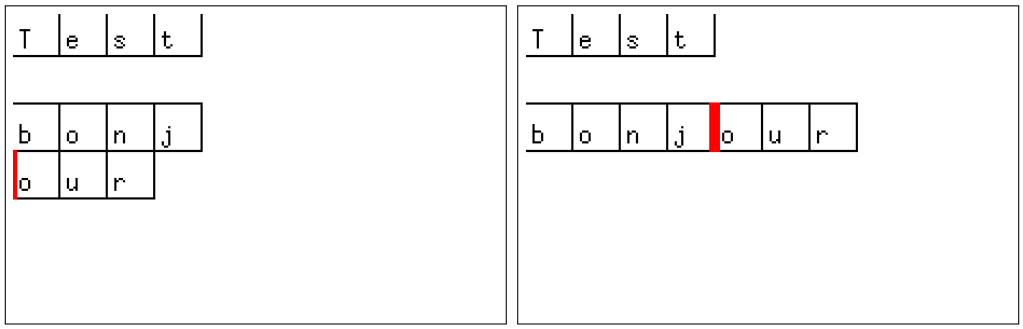

FIGURE 4.5 – Backspace et fusion de deux lignes

Problèmes rencontrés et limites du programme

5.1. Problèmes rencontrés

Le principal problème a été rencontré lors de mes tests concernant le modifications de position du curseur. En effet, lors des déplacements verticaux, le curseur ne se positionnait pas à l'endroit souhaité. J'ai donc compris que l'affichage graphique de ce dernier ne dépendait pas que du paramètre `pos` du zipper, mais aussi de la longueur de `before`. J'ai donc résolu ce problème à l'aide de la fonction `take`, utilisée dans `move_up` et `move_down`.

5.2. Limiges du programme

L'éditeur présente certaines limitations fonctionnelles.

Tout d'abord, il ne gère pas le retour automatique à la ligne : si l'utilisateur saisit plus de caractères que la largeur de la fenêtre ne peut en afficher, ceux-ci continuent d'être ajoutés sur la même ligne sans rupture visuelle. Cela nuit à la lisibilité des longues lignes et pourrait être amélioré par une logique de retour à la ligne automatique ou un affichage en défilement horizontal.

Par ailleurs, le programme ne prend pas en charge des fonctionnalités telles que le *copier/coller* ou la *sélection* de texte. Ces opérations impliqueraient sans doute de concevoir un système de mémoire tampon temporaire, ainsi qu'une gestion visuelle de la sélection.

Enfin, aucune fonctionnalité de *sauvegarde* ou de *chargement de fichier* n'a été intégrée. Le texte entré par l'utilisateur est donc perdu à la fermeture de la fenêtre.

Conclusion

Ce projet m'a permis de concevoir un petit éditeur de texte fonctionnel en OCaml. Grâce à l'utilisation des zippers, j'ai pu manipuler le texte de manière simple et efficace, en gardant une logique claire autour du curseur.

Même si l'éditeur reste basique, il fonctionne bien et pourrait être amélioré à l'avenir en ajoutant des options comme la sauvegarde de fichiers, le copier-coller ou encore une meilleure interface graphique (sélection de texte..).