# Projet individuel d'algorithmique-programmation
# AP1 :
# groupe 3.2

octobre 2010

## 1 Informations générales

### 1.1 Travail à rendre

Le projet est à réaliser en OCaml **individuellement**. Il sera accompagné d'un **dossier** contenant impérativement la description des choix faits, la description des types et des fonctions. Pour chaque fonction, on donnera impérativement l'interface complète (dans le code en commentaire et dans le rapport pour les fonctions présentées).

Même si le sujet est décomposé en questions, il est possible qu'une question se résolve par l'écriture d'une ou plusieurs fonctions intermédiaires. Celles-ci doivent comporter une interface également.

Le dossier fournira également des cas de tests accompagnés des résultats attendus et retournés.

Sur le site du cours figure un petit document sur ce que l'on attend dans un rapport. Consultez-le !

### 1.2 Calendrier et procédure de remise

Le projet (dossier + copie du listing de code) est à rendre au secrétariat **le 22 novembre à 16h au plus tard**. Le numéro du groupe, ainsi que le nom du chargé de TD, devront figurer en gros, et en rouge sur la page de garde.

Le fichier .ml contenant votre code devra être déposé électroniquement au plus tard **le 22 novembre à minuit**.

Les soutenances seront organisées dés la semaine suivante. Il vous faudra consulter les panneaux d'affichage et votre courrier electronique pour obtenir l'ordre de passage.

Enfin n'attendez pas pour vous mettre au travail ! Un projet se travaille dès la remise du sujet afin d'avoir le temps de laisser murir la solution et de poser des questions au client (dans votre cas, votre chargé de TP).

### 1.3 Procédure de dépot

Pour déposer le code du projet, consultez les informations sur les machines de l'école, tout figure.

Ne vous inquiétez pas, l'interface fabrique un nom de projet déposé à partir de votre login. Votre projet ne sera pas confondu avec celui d'un autre.

Indiquez quand même en commentaire dans votre fichier de code Ocaml votre nom et votre groupe. Ce sera plus facile pour le correcteur.

Vous pouvez déposer successivement plusieurs versions, le dernier dépot écrase le précédent, seul le dernier dépot est pris en compte

Enfin tout cela peut se faire de l'extérieur.

Le code à déposer est un fichier.ml commenté **avec les interfaces des fonctions**. **Le fichier doit pouvoir être compilé (sans aucune intervention humaine - lecture du buffer complet) sur les machines Yaka de l'école**, n'oubliez pas de vérifier que tout fonctionne sur les machines de l'école avant de le dép oser - si vous l'avez développé sous un autre système d'exploitation.

## 2 Énoncé du projet : Interprétation et optimisation d'un langage de formatage de texte

Un langage de formatage de texte permet de décrire comment un texte doit être rendu (police, graisse, corps, soulignement, couleur, ...). On peut par exemple citer HTML, LaTeX, (t—n)roff, RTF, etc. On se propose d'écrire un programme utilisant la librairie graphique d'ocaml pour interpréter un tel langage. Ensuite, nous chercherons à optimiser des expressions de ce langage, c'est-à-dire que nous chercherons à réduire la taille des expressions tout en conservant la même interprétation.

### 2.1 Le langage BIB

Le langage BIB (*BIB isn't BBCode*[1]) permet d'ajouter des informations de rendu à un texte à l'aide de balises. Il est possible de :

- mettre un texte en gras `[b]texte gras[/b]`
- mettre un texte en italique `[i]texte en italique[/i]`
- mettre un texte en valeur `[em]texte mis en valeur[/em]`, c'est-à-dire passer en italique si le texte autour est en romain, et en romain si le texte autour est en italique ;
- changer de taille `[+]texte agrandi[/+]` ou `[-]texte plus petit[/-]`

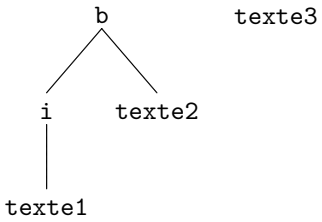
1. Ce langage a été inventé pour ce projet et n'est en aucun cas lié à d'autres langages qui porteraient le même nom.

- souligner `[u]texte souligné[/u]`
- passer en chasse fixe `[tt]texte en chasse fixe[/tt]`
- mettre en couleur `[color=`$c$`]texte coloré[/color]` où $c$ peut être
  - `black` : noir ;
  - `white` : blanc ;
  - `red` : rouge ;
  - `green` : vert ;
  - `blue` : bleu ;
  - `yellow` : jaune ;
  - `cyan` : bleu clair ;
  - `magenta` : magenta.
- repasser en texte romain, non gras, de taille normale, non souligné `[pl]texte normal[/pl]`

Les balises peuvent être imbriquées, par exemple `[b][u]texte gras et souligné[/u][b]` Toutefois, le niveau de soulignement ne dépasse jamais 3, et on n'augmente et ne diminue les tailles jamais plus de 5 fois par rapport à la taille normale.

Une balise peut en contenir plusieurs, par exemple
`[b][u]gras souligné[/u][i]gras italique[/i]gras romain[/b]`

On distinguera le code BIB, c'est-à-dire les suites de caractères décrivant un objet BIB, des expressions BIB, c'est-à-dire une forêt décrivant un objet BIB. Par exemple, le code `[b][i]texte1[/i]texte2[/b]texte3` ne devra pas être confondu avec l'expression




même si les deux représentent le même objet. (On parle de syntaxe concrète et de syntaxe abstraite.)

Par conséquent, on considérera des listes d'expressions du langage BIB qui seront elles-mêmes constituées :

- soit d'une chaîne de caractères ;
- soit d'une balise `b` encadrant une liste d'expressions BIB ;
- soit d'une balise `color` associée à une couleur et encadrant une liste d'expression BIB ;
- ...

Pour ce projet, on définira un type ocaml `bib` correspondant aux expressions du langage BIB, mais on ne cherchera pas à savoir comment passer du code BIB à une représentation de ce code en ocaml.

# 3 La bibliothèque graphique d'ocaml

Ocaml est fourni avec une bibliothèque graphique, c'est-à-dire un ensemble de fonctions basiques qui permettent de faire des dessins. La bibliothèque n'est pas chargée par défaut, par conséquent dans l'interpréteur ocaml il faut taper

```
#load "graphics.cma";;
```

(le `#` doit être tapé) tandis que pour compiler un fichier `toto.ml` utilisant la bibliothèque graphique il faut faire

```
ocamlc -o toto graphics.cma toto.ml
```

L'ensemble des primitives graphiques est décrit dans le manuel d'ocaml `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`, nous nous contentons de décrire celles qui nous seront utiles.

- `open_graph ""` permet d'ouvrir une fenêtre dans laquelle seront effectués les dessins ; il est indispensable d'exécuter cette commande avant d'appeler les fonctions de dessin ;
- `clear_graph ()` permet d'effacer la fenêtre de dessin ;
- `set_color c` permet de changer la couleur courante ; `c` peut être `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.

Les dessins sont fait par rapport à un point courant. Lors de l'ouverture de la fenêtre avec `open_graph`, le point courant est le point en bas à gauche, de coordonnées (0,0).

- `moveto x y` place le point courant à (x,y) ;
- `rmoveto x y` décale le point courant de x vers la droite et de y vers le haut (x et y peuvent être négatifs) ;
- `rlineto x y` trace une ligne partant du point courant et allant de x vers la gauche et de y vers le haut ; le point courant devient la fin de la ligne ;
- `draw_string ch` dessine la chaîne de caractères ch en mettant le point courant comme coin inférieur gauche ; le point courant devient le coin inférieur droit ;
- `text_size ch` retourne un couple donnant la taille qu'aurait la chaîne de caractères ch si elle était dessinée avec la fonte courante ;
- `set_font ch` change la fonte courante pour dessiner les chaînes ; l'interprétation de ch dépend de l'implémentation ; sous les machine Unix de l'école, ch est une description de police X11R6 ; sans rentrer dans les détails, nous serons intéressé par deux familles de fontes, décrites à l'aide de

  `"-*-fmly-wght-slant-*-*-pxlsz-*-*-*-*-*-*-*"`

  où `fmly` sera soit `helvetica` (texte normal), soit `courier` (texte en chasse fixe) ; `wght` sera soit `bold` (gras), soit `medium` ; `slant` sera soit `i` (italique) soit `r` (romain) ; `pxlsz` sera 8, 10, 11, 12, 14, 17, 18, 20, 24, 25 ou 34. La taille par défaut sera donc 17.

  Pour savoir à quelle fonte correspond une description, il est possible d'utiliser l'outil `xfontsel` sous Unix.

# 4 Optimisations

Il est parfois possible d'optimiser des expressions BIB, c'est-à-dire de trouver des expressions plus petites avec la même sémantique. Par exemple, au lieu de `[b][b]texte[/b][/b]` on peut préférer `[b]texte[/b]`. Dans ce projet, on voudra traiter au moins les optimisations suivantes :

- élimination des redondances, par exemple `[b][i][b]texte[/b][/i][/b]` devient `[b][i]texte[/i][/b]` ;
- réduction des changements de taille, par exemple `[+][b][-]texte[/-][/b][/+]` devient `[b]texte[/b]` ;
- jointure de style, par exemple `[b][tt]texte1[/tt][/b][i][b]texte2[/b][/i]` devient `[b][tt]texte1[/tt][i]texte2[/i][/b]` ;
- raccourci utilisant `pl`, par exemple `[b][i][tt][+]texte1[/+][/tt][/i][/b]texte2[tt][i][+][b]texte3[/b][/+][/i][/tt]` devient `[b][i][tt][+]texte1[pl]texte2[/pl]texte3[/+][/tt][/i][/b]`.

Vous pourrez bien entendu considérer d'autres optimisations.

# 5 Travail à fournir

1. Définir un type ocaml `bib` pour les expressions BIB.
2. Écrire une fonction qui prend une expression de type `bib` et qui renvoie la chaîne de caractères de son code BIB.
3. Écrire une fonction qui interprète les expressions de type `bib`, c'est-à-dire qui utilise la bibliothèque Graphics pour mettre en forme le texte représenté par une expression BIB.
4. Écrire des fonctions qui optimisent une expression de type `bib`. Pour chacune des optimisations proposées, on montrera :
   - pourquoi l'optimisation est correcte (c'est-à-dire qu'elle préserve la sémantique) ;
   - pourquoi l'optimisation est utile (c'est-à-dire qu'elle diminue la taille de l'expression BIB, cette taille étant son nombre de nœuds si on la représente sous forme de forêt).