Mai, 2013
Luka Nerima
Informatique pour les sciences humaines,
MA6 Nouvelles Technologies de l'Information
et de la Communication (NTIC)

Projet NTIC : Sortie graphique d'arbre d'analyse pour le service Web Fips

Samuel Constantino
ferrese0@etu.unige.ch

I - Introduction

L'objectif de ce projet est de créer une application web permettant de représenter sous forme d'arbre, l'analyse syntaxique d'une phrase donnée. Cette application se sert de l'analyseur multilingue Fips - conçu par le Laboratoire d'Analyse et de Technologie du Langage (LATL) - qui interprète la structure syntaxique de phrases.

Notre application web (appelée FipsTree) transforme les constituants analysés par Fips en nœuds d'un arbre et retourne, une fois les calculs nécessaires finis, cet arbre en une image vectorielle dans le navigateur de l'utilisateur. Cette tâche implique une bonne utilisation de la nature récursive des arbres, ainsi qu'une bonne compréhension de l'analyse donnée par le service Fips.

Bien qu'il existe d'autres applications en ligne permettant de faire cela - par exemple, ce projet assez similaire[1], ou encore ce logiciel de création d'arbre[2] - la focalisation ici est d'essayer de créer des arbres où la disposition des nœuds soient assez esthétiques : en prenant comme axiome principal que toutes les branches des arbres seront placées parallèlement les unes aux autres.

Dans ce rapport, nous allons expliquer tout d'abord l'idée principale (interface, technique) du projet ; puis nous allons voir comment fonctionnent les algorithmes du programme ; ensuite, nous allons discuter des différents aspects qui tiennent au dessin de l'arbre ; et finalement, nous allons parler des différents résultats obtenus par cette application et des difficultés et limites rencontrées durant la conception.

[1] http://ironcreek.net/phpsyntaxtree/ - application en php également, mais qui demande à l'utilisateur une phrase déjà parsée.
[2] http://www.ece.ubc.ca/~donaldd/treeform.htm - logiciel pour créer des arbres graphiques seulement, sans l'analyse de phrase.

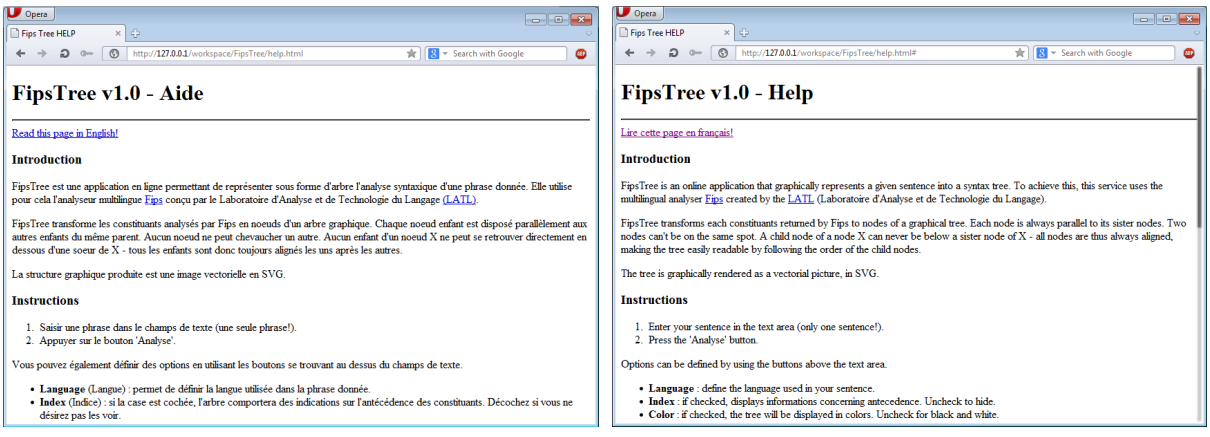
II - Idée

Interface :

FipsTree se présente sous la forme d'une très simple page HTML comme suit :

(l'interface telle qu'elle apparaît sous Opera)

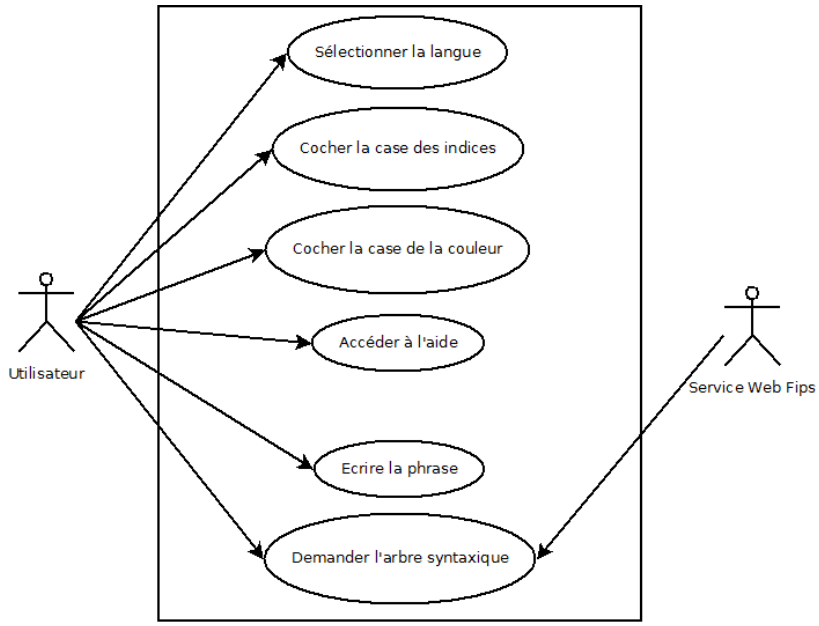
L'utilisateur peut faire les actions suivantes : définir les options quant à la langue, l'affichage des indices (si la case est cochée, l'arbre comportera des indications sur l'antécédence des constituants), la couleur (si la case est cochée, les différentes parties de l'arbre seront colorées - sinon, l'image sera en noir et blanc) et accéder à l'aide.



(la page d'aide est accessible en français et en anglais)

Finalement, l'utilisateur peut saisir sa phrase dans le champ de texte et appuyer sur le bouton 'Analyse' pour générer l'arbre syntaxique correspondant à sa phrase.

USE cases :



Ce schéma UML décrit les actions que l'utilisateur peut faire. Une fois qu'il demande à visualiser l'arbre, FipsTree fait appel au service Fips externe pour analyser la structure de la phrase.

Outils choisis :

Les technologies suivantes ont été utilisées pour réaliser ce projet : PHP, pour la partie algorithmique qui conçoit, calcule et gère l'entrée et la sortie du programme ; et SVG, pour représenter graphiquement l'arbre.

J'ai choisi PHP puisque nous avons affaire ici à un service web, que nous devons traiter une requête vers Fips, ainsi que traiter la réponse à cette requête (le traitement du XML), et finalement écrire du SVG (retourner du HTML/XML en PHP étant très simple). Un autre grand avantage constaté pendant la conception du code a été l'aspect orienté objet de PHP - utile pour la structure de données utilisée - et surtout la façon dont les instances d'objets sont appelées par références (nos structures contiennent de nombreux liens vers les nœuds parents, enfants, les nœuds aux extrémités…). SVG semblait être le choix le plus simple pour cette tâche car nos arbres sont simplement du texte et quelques lignes - ce qui est très simple à faire avec ce langage basé sur des balises XML. Au départ, j'avais également imaginé utiliser Javascript pour traiter les chevauchements. Cependant, il s'est avéré plus rapide et efficace de tout faire en PHP en utilisant au mieux la structure d'objets citée plus haut.

III - Fonctionnement de l'algorithme

Plan de l'algorithme :

L'algorithme de FipsTree est (plus ou moins) organisé en suivant l'architecture MVC (Modèle-vue-contrôleur), c'est-à-dire qu'il est divisé en parties selon leur usage dans le code - la partie du modèle du code (sa structure ; ici notre classe Noeud), la partie du rendu visuel (l'affichage en SVG) et la partie qui dirige toutes ces opérations (le script principal appelé par l'appui du bouton).

Le contrôleur (fipstree.php), une fois que l'utilisateur l'appelle, va faire les actions suivantes : tout d'abord, il appelle le script qui communique avec le service Fips ; une fois qu'il obtient le résultat de cette requête, il appelle le script qui transforme ce retour (un fichier XML) en une structure d'objets de la classe Nœud ; puis il appelle le script qui va calculer pour cette structure s'il y a des chevauchements et repositionner l'arbre tant qu'il y en a ; finalement, il va dessiner l'arbre en SVG en appelant le script qui s'occupe de cela.

Communication avec le service web Fips :

Le script qui s'occupe de créer la requête vers Fips et de récupérer la réponse a été écrit par Kamel Nebhi. Ce code est assez simple. Il récupère d'abord la saisie de l'utilisateur dans le formulaire (le texte et l'option de langue) :

```
$phrase = $_POST['in']; // le champs de text

$langue = $_POST['ln']; //langue

$application = "Xml";

//definition des parametres de la requete
$data = array('in'=>$phrase,
                    'ap'=>$application,
                   'ln'=>$langue);

$requete = http_build_query($data);
```

Puis il utilise l'extension de PHP nommé php_Curl qui remplit un formulaire HMTL (celui de Fips) :

```
// initialisation curl
$ch = curl_init();
// parametres
curl_setopt($ch, CURLOPT_URL, $url); // url
```

Et enfin, il récupère la réponse dans une variable :

```
$reponse = curl_exec($ch);
```

Parcours du XML :

La réponse récupérée est un objet contenant un fichier XML. Fips permet visualiser l'analyse d'une phrase sous plusieurs formats. Par exemple, pour la phrase :

Le chat mange la souris

Fips peut nous donner le résultat sous la forme d'une phrase « parsée » à l'aide d'accolades :

[TP[DP Le [NP chat]] mange [VP [DP une [NP souris]]]]

Sous la forme d'un fichier XML :

```
<!DOCTYPE LATLPARSE SYSTEM "http://www.latl.unige.ch/xml/latlparse.dtd">
<LATLPARSE xml:lang="fr">
  <ANALYSIS complete="yes" score="-44">
    <PROJ cat="TP">
      <PROJ cat="DP">
        <HEAD cathead="D" cat="D" lexeme="le" gender="masc" number="sin">
          le
        </HEAD>
        <PROJ cat="NP">
          <PUNC key="space"/>
          <HEAD cathead="N" cat="N" lexeme="chat" gender="masc" number="sin">
            chat
          </HEAD>
        </PROJ>
      </PROJ>
      <BAR cat="T" colorGramm="pred" tree="GV">
        <PUNC key="space"/>
        <HEAD cathead="T" cat="V" lexeme="manger" number="sin" person="3">
          mange
        </HEAD>
        <PROJ cat="VP">
          <PROJ cat="DP">
            <PUNC key="space"/>
            <HEAD cathead="D" cat="D" lexeme="un" gender="fem" number="sin">
```

Ou encore en XMLTei (du XML avec d'autres types de balises) ou bien sortir des informations sur les constituants à l'aide du *tagger*.

Pour FipsTree, j'ai décidé d'utiliser la sortie en XML de Fips comme base, car c'est celle qui propose le plus d'informations par rapport à la phrase (par exemple, des informations sur les antécédents pour les indices, des informations pour savoir si l'analyse est complète, etc…).

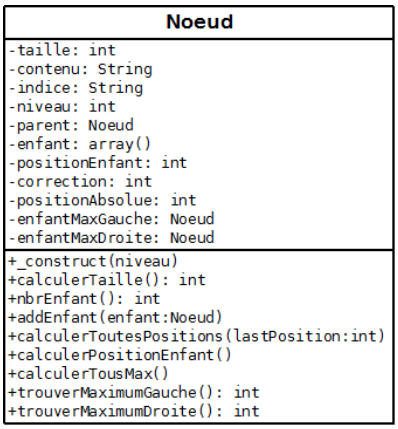
Le fichier XML rendu par Fips est composé de la façon suivante : il comprend d'abord deux balises entrantes **<LATLPARSE>** (la balise principale), puis **<ANALYSIS>** (la phrase - c'est ici qu'on sait si l'analyse est complète). Ensuite, il contient les deux types de balises enchâssées **<PROJ>** (les projections) et **<BAR>** (les projections intermédiaires) ; et enfin **<HEAD>** qui contient le mot.

FipsTree traite ces balises dans la fonction récursive ParcourirNoeud() du script load.php. S'il est dans une balise <PROJ> du XML, alors il garde la valeur de l'attribut « cat » comme contenu du nœud (le nom de la projection), s'il est dans une balise <HEAD>, alors c'est ce qu'il y a entre les balises qu'il garde en contenu (le mot). Dans les autres cas, il ne garde rien, car on omet les projections intermédiaires du schéma X-Bar. A chaque balise, une fois les traitements faits, il va relancer cette même méthode dans les enfants de chaque nœud.

La classe Nœud :

En traversant le fichier XML, l'algorithme enregistre toutes ces informations dans des instances de la classe Nœud. C'est sur ces objets que nous allons faire tous les calculs concernant les chevauchements, et que nous allons dessiner le SVG depuis.

Cette classe se présente de la façon suivante :



(j'ai omis les setters et les getters par souci de place)

Les attributs de cette classe sont des informations sur le contenu (le mot ou le nom de la projection), l'indice, le niveau (la racine de l'arbre se trouve au niveau 0, puis chaque étage est numéroté 1, 2, 3…), les enfants (un tableau où chaque élément est un Nœud) et d'autres Nœud que nous allons expliquer plus loin.

Pour ajouter un enfant à un Nœud, l'opération est extrêmement simple en PHP. Dans ce langage, la taille des tableaux n'a pas besoin d'être spécifiée et on peut ajouter des éléments dynamiquement de la façon suivante :

```
$this->enfant[] = $enfant;
```

Un autre attribut important est la taille. Celle-ci est mesurée par la fonction CalculerTaille() qui retourne la taille en pixel pour son contenu, telle qu'il apparaîtra sous SVG. Pour cela, j'ai utilisé la fonction prédéfinie imagettfbbox() qui prend en paramètre des informations concernant la taille et le type de la police utilisée.

```
// retourne la taille telle que calculée par SVG
public function calculerTaille(){
        //crée une 'boite' autour d'un contenu - les dimensions de la boite correspondent à la taille en pix
        $box = imagettfbbox(TAILLE_POLICE, 0, NOM_POLICE_FICHIER, $this->getContenu());

        //position du coin droit - position du coin gauche (peut être négatif)
        return abs($box[2] - $box[0]);
}
```

CalculerToutesPositions() est une fonction récursive qui pour chaque nœud calcule d'abord les positions des enfants (voir partie sur le dessin pour l'explication) avec CalculerPositionEnfant(), puis calcule sa position absolue dans l'arbre et s'appelle récursivement pour chacun des enfants de ce nœud (ainsi, si une modification est appliquée à ce noeud, on recalcule toutes les positions de ses enfants).

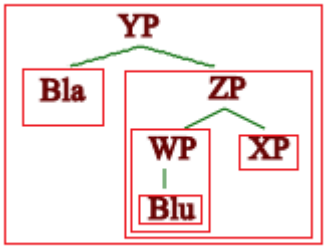
Les fonctions récursives calculerTousMax(), trouverMaximumGauche() et trouverMaximumDroite() servent, après chaque modification, à vérifier quels enfants d'un nœud se trouvent aux extrémités vers la gauche et vers la droite (pour le calcul des chevauchement) - en commençant par chercher les maximums pour les enfants les plus éloignés du nœud demandé.

IV - Dessin de l'arbre :

Concepts principaux :

Les règles pour le dessin de l'arbre, imposées au départ pour ce projet, était les suivantes : chaque nœud enfant est disposé parallèlement aux autres enfants du même parent. Aucun nœud ne peut chevaucher un autre. Aucun enfant d'un nœud X ne peut se retrouver directement en dessous d'une sœur de X - tous les enfants sont donc toujours alignés les uns après les autres (en « escalier »).

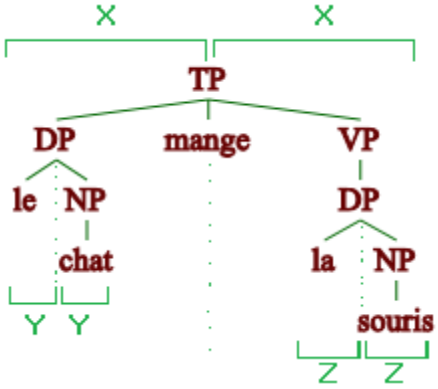
Pour réaliser cela, la récursivité - qui est le propre des nœuds d'un arbre - a dû être utilisée à bonne escient. Chaque nœud de notre fichier SVG sera en lui-même un SVG interne (comme le montre la figure ci-contre). En entourant ainsi les nœuds de balises <SVG>, on n'a pas besoin de savoir exactement où chaque nœud se trouve, mais seulement où ils se trouvent par rapport à leur parent.



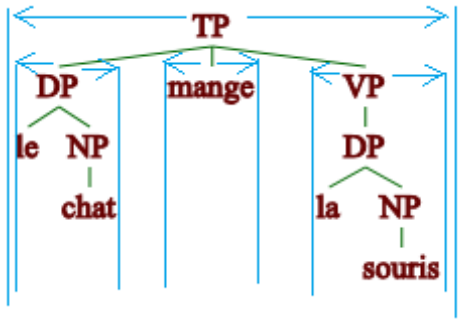

```
<svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" overflow="visible">

<svg x="20" y="20" overflow="visible"> <!--Groupe 1-->

<text x="20" y="20" style="stroke: #660000; fill: #660000">YP</text> <!--le texte-->
<line x1="28" y1="25" x2="8" y2="35" style="stroke: #006600;"/> <!--les lignes-->
<line x1="28" y1="25" x2="48" y2="35" style="stroke: #006600;"/>

<svg x="30" y="30" overflow="visible"><!--Groupe 2-->
<text x="20" y="20" style="stroke: #660000; fill: #660000">ZP</text> <!--le texte-->
<line x1="28" y1="25" x2="8" y2="35" style="stroke: #006600;"/>    …
```

(Exemple d'un morceau d'un fichier SVG créé par notre algorithme)

Nous utilisons donc les positions relatives (et non absolues - donc par rapport à l'ensemble du document) pour positionner les enfants. Pour que chaque enfant se trouve de façon parallèle aux autres enfants du même parent, il suffit de réduire à une seule valeur la distance qui les sépare, comme l'exemplifie le schéma suivant :



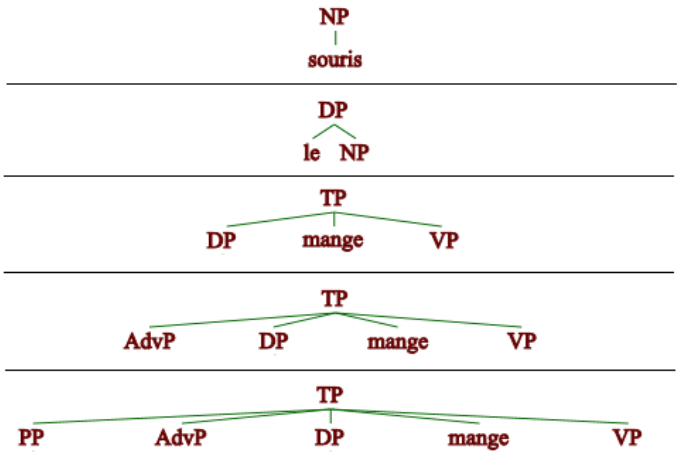
(Par exemple, tous les enfants de 'TP' sont placés à X distance les uns des autres)



Pour les chevauchements et l'alignement (aucun nœud enfant ne peut se trouver sous un nœud du même parent), cependant, nous utilisons les positions absolues cette fois-ci pour faire les modifications. Il suffit de comparer les distances maximums qu'atteignent les enfants d'un nœud avec les distances maximums du nœud suivant. Par exemple, dans la figure suivante, entre 'DP' et 'mange', on compare à quelle position est l'extrémité à droite de l'enfant le plus à droite ('chat') de 'DP', et la position à gauche de 'mange'.

Positionnement de multiples enfants :

Positionner les enfants d'un nœud se fait donc avec une seule valeur. Comment calculer alors avec cette valeur le positionnement en parallèle avec deux enfants, trois enfants, quatre, etc… ? Observez l'exemple suivant :



Comme vous pouvez le remarquer ici, il y a deux cas possible : soit nous sommes face à une formation avec un nombre pair d'enfant, soit à un nombre impair. Pour les nombres pairs, il n'y a jamais de nœud au milieu ; pour les nombres impairs, il y en a toujours un (y compris la formation avec un seul enfant). Donc pour savoir où placer l'enfant, il suffit de multiplier par un indice de distance par rapport au centre, la valeur de positionnement des enfants. Si nous sommes face à un nombre impair, il faut alors ajouter une fois cette valeur (pour prendre en compte le nœud du centre) ; sinon, il ne faut l'ajouter qu'une demie fois par enfant autour du centre (pour avoir une fois cette distance au milieu).

Calcul des chevauchements :

Le calcul des chevauchements se fait en suivant le principe décrit plus haut. Tout d'abord il faut calculer les positions absolues de chaque nœud. Pour cela, une simulation de la fonction de positionnement de l'arbre décrite dans le paragraphe précédent est lancée (la fonction calculerToutesPositions() simule la fonction GenerateSVGNode()) pour savoir où doivent se trouver les nœuds. Ensuite, récursivement, on compare pour chaque deux enfants (un enfant et le enfant qui le suit - soit à la position i+1) les positions de leurs enfants se trouvant aux le plus loin à gauche et à droite.

Donc pour l'enfant à gauche, on veut savoir la position à droite maximum de ses enfants, et pour le nœud à droite on veut la position à gauche maximum de ses enfants :

```
$maxNoeudGauche = $noeud->getEnfant($i)->getMaxDroite();
$maxNoeudDroite = $noeud->getEnfant($i+1)->getMaxGauche();

$maxGauche = $maxNoeudGauche->getPositionAbsolue() + $maxNoeudGauche->getTaille() + DEFAUT;
$maxDroite = $maxNoeudDroite->getPositionAbsolue();
```

S'il y a chevauchement (si ces deux distances se croisent), alors on modifie la distance entre les enfants avec la différence et on recalcule le tout (y compris les maximums).

Post-traitement :

Une fois tous les chevauchements éliminés, on va centrer l'arbre - car on utilise pour calculer les positions absolues une position constante arbitraire - en prenant la distance maximum à gauche pour la racine et soustraire cette valeur à la position arbitraire.

Puis on calcule les dimensions totales du document pour pouvoir se déplacer (avec les flèches de navigation) dans le SVG si le document est plus grand que la fenêtre. En effet, avec SVG, il n'est pas nécessaire de définir des dimensions pour les documents. Cependant si on ne le fait pas, alors on risque de dépasser la taille fenêtre de l'utilisateur et on ne pourra pas voir ce qui se trouve au-delà. Pour calculer ces valeurs, on regarde les maximums gauche/droite et on calcule la hauteur selon le niveau le plus bas obtenu.

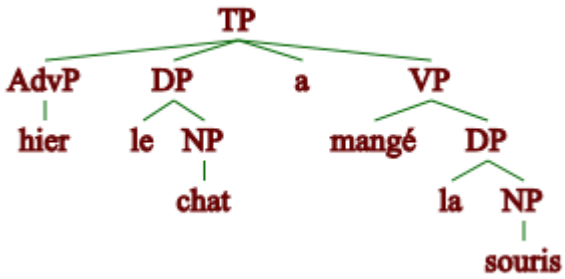
Parcours et transformation des objets Nœud en SVG :

Une fois toutes ces données calculées, on peut transformer la structure de Nœud en SVG. Deux fonctions font cela : la première, createSVG() crée les balises entrante et sortante du document et appelle pour tous les Nœud la seconde, CreateSVGNode(). Celle-ci prend en paramètre l'orientation du Nœud (avant ou après le centre) et multiplie cette orientation (-1 à gauche, 0 au milieu, et 1 à droite) par la distance entre les enfants. Il suit ensuite l'algorithme décrit précédemment pour les placer tous en parallèle selon le nombre d'enfant.

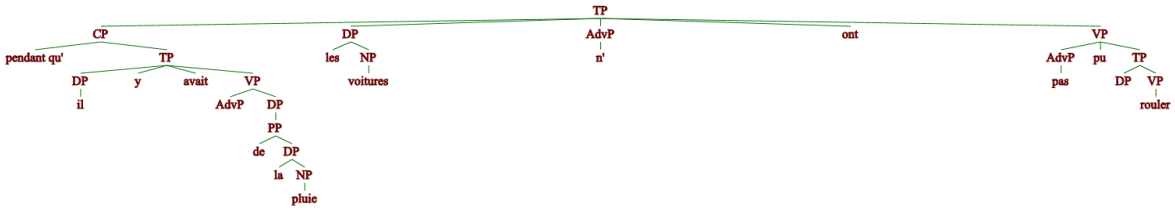
V - Tests, résultats et limites :

Le problème des parallèles :

Dans l'ensemble, les arbres générés par FipsTree sont assez esthétiques - c'est ce que nous pouvions attendre grâce au positionnement en parallèle :



Cependant, dans les cas où certains constituants sont plus longs que d'autres, ce type de positionnement n'est pas très adapté, comme nous pouvons le voir ici :



Bien que cela soit dommage, en général les arbres sont plutôt bons et la technique des parallèles semble être une bonne idée. Il serait intéressant néanmoins d'imaginer un moyen d'éviter ce résultat en le prenant en compte d'une certaine manière - bien que cela sorte de la prémisse de ce projet - dans une hypothétique version future.

Difficulté des chevauchements :

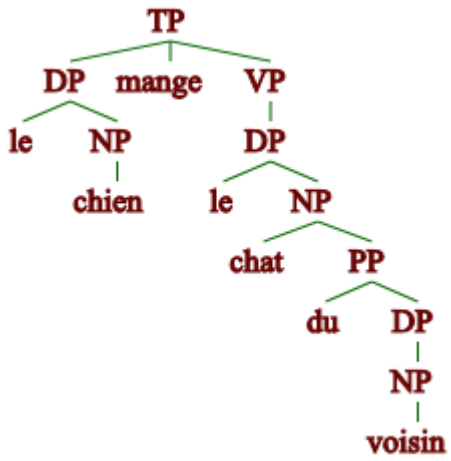
Une première itération de l'algorithme de recherche de chevauchement consistait à vérifier les chevauchements entre chaque nœud par étage uniquement. Ainsi dans la phrase suivante, on créait un tableau à deux dimensions (les étages, puis chaque nœud) où l'on testait s'il y avait

des chevauchements par étage. Cependant, cette méthode n'était pas satisfaisante car elle produisait l'effet suivant :

TP

DP mange VP
---
le NP DP
---
chien le NP
---
chat PP
---
du DP

NP
---
voisin



(le tableau par étage et son rendu avec cette méthode)

Dans cet arbre, l'effet « escalier » n'est pas présent : par exemple, 'chien' et sa projection se trouvent en dessous de 'mange' - pareil pour 'chat' qui se trouve en dessous de 'le'. Il fallait donc créer une autre fonction pour vérifier cela. Comme montré précédemment, la vérification de cet effet implique le fait que les nœuds ne se chevauchent pas. La méthode par étage est donc devenue obsolète et a été retirée du produit final.

Compatibilité et taille du texte :

La fonction décrite précédemment imagettfbbox() - qui nous aide à calculer la taille des mots en SVG - prend en paramètre la police utilisée pour rendre le texte. Afin que cette taille soit réellement la taille que l'utilisateur voit à l'écran, j'ai inclus avec le code sur le serveur le fichier TTF de la police utilisée - ainsi, si un utilisateur ne la possède pas, elle est chargée par SVG sur le serveur lors de l'affichage et on s'assure de la sorte que SVG n'utilise pas une autre police.

Traitement des erreurs :

Plusieurs erreurs sont possibles au cours de ce programme. Dans ce cas, un message pour l'utilisateur s'affiche. Par exemple, quand Fips n'arrive pas à comprendre la phrase, le message suivant apparaît :

Cependant, des erreurs inconnues sont également possibles. Il faut alors se référer à la page d'aide dans ces cas-là.

VI - Conclusion

Dans l'ensemble, ce projet - personnellement - me semble assez réussi. Les objectifs annoncés au départ (produire un arbre où les enfants seraient positionnés parallèlement et alignés) ont été remplis et le rendu final est plaisant.

Ce projet m'a permis de mettre en pratique les notions théoriques du premier semestre et surtout de maitriser le PHP et SVG. La conception de l'algorithme qui place un nombre indéterminé d'enfants a été également un *challenge* logiquo-géométrique intéressant à mener.

En ce qui concerne les améliorations que l'on pourrait encore apporter à ce programme, on pourrait par exemple souhaiter améliorer la notation des indices. Comme cette partie a été l'une des dernières réalisées, je n'ai pas eu le temps d'avoir un avis expert sur la qualité de celle-ci. L'algorithme retranscrit exactement la position où l'indice se trouve dans l'arbre selon le XML donné par Fips, mais on pourrait par exemple imaginer que les indices n'apparaissent qu'aux extrémités de l'arbre. Néanmoins, si l'utilisateur le veut, il a la possibilité de ne pas les faire afficher.

Quant aux développements futurs envisageables, on peut facilement imaginer une option d'interaction (possible en SVG) pour réduire des branchements complets de l'arbre en un seul nœud (résumé avec un triangle au lieu d'un trait), ou encore, comme il était question au départ du projet d'avoir l'option d'enregistrer l'arbre en PDF. L'arbre ne peut être enregistré qu'en format SVG pour le moment. Cependant, ce format est facilement convertissable, comme décrit dans la rubrique d'aide (et sous IE 9, il est même possible de faire un click droit et d'enregistrer l'arbre en d'autres formats).