UNIVERSITÉ CATHOLIQUE DE LILLE
L2 SCIENCES DU NUMÉRIQUES

Algorithmes avancés

Projet NanoML

Charles Iacopino - Guillemet Steven

2023 - 2024

1. Introduction

1.1 Contexte

Dans le cadre de notre cours d'Algorithme avancé, notre projet consiste à développer un programme pour analyser syntaxiquement des fichiers au format NanoML, inspiré de HTML. Ce travail comprend plusieurs étapes cruciales débutant par :

- L’analyse syntaxique pour assurer la conformité à une grammaire prédéfinie.
- Suivi de la construction d'un arbre n-aire représentant la structure du document NanoML.
- L’illustration de cet arbre via un affichage spécifique.

1.2 Objectifs

Après avoir eu une discussion approfondie entre nous sur les objectifs que nous voulons nous fixer pour ce projet NanoML, nous avons convenu de prioriser l’analyse syntaxique comme première étape essentielle, car étant le « travail minimal » à effectuer. L'objectif clé de notre projet est de concevoir un outil dédié à l'analyse syntaxique, assurant leur conformité au format spécifié. Cette première étape constitue le socle sur lequel repose l'ensemble du projet. Une fois cette base validée, nous envisageons d'étendre ses capacités à la construction d'un arbre n-aire. Cet arbre servira à mémoriser et représenter fidèlement la structure du document analysé. S’ensuit la visualisation de cette structure se concrétisera par un parcours récursif de l'arbre, affichant chaque élément dans des cadres soigneusement délimités pour garantir clarté et cohérence, sans débordement.

1.3 L’équipe et la méthode de travail

Notre équipe, composée de Charles Iacopino et de Guillemet Steven. Nous avons commencé le projet NanoML très tôt pour éviter les problèmes du premier semestre.

Initialement, nous avons minutieusement analysé le projet, délimitant clairement nos rôles. Suite à cela nous avons réalisé une arborescence qui servira à mieux représenter le projet. Pour réaliser ces différentes activités, nous avons opté pour l'utilisation d'un projet hébergé sur GitHub, lequel a été régulièrement actualisé via l'emploi de branches distinctes afin d'éviter tout effet sur la branche principale, avec une fusion effectuée seulement en conclusion du projet. Nos emplois du temps présentent une certaine flexibilité, cependant, grâce à notre disponibilité mutuelle fréquente, nous avons l'habitude de communiquer souvent via Discord. Nous convenons également de moments dédiés au cotravail durant les week-ends ou pendant les périodes creuses de la semaine, comme les mercredis après-midi après 12 heures.

1.4 Structure du rapport

Le rapport se divise en plusieurs parties clés, reflétant les différentes phases du projet :

- Introduction au projet et aux objectifs.
- Description détaillée de la grammaire NanoML et des règles syntaxiques.
- Méthodologie appliquée pour l'analyse syntaxique et la construction de l'arbre n-aire.
- Fonctionnement du programme.
- Discussion sur la mise en œuvre de la visualisation de l'arbre et les défis rencontrés.
- Analyse des résultats obtenus et des améliorations potentielles.
- Conclusion générale et perspectives d'évolution du projet.

2. Description détaillée de la grammaire NanoML et des règles syntaxiques.

Le choix de notre analyse combine des fonctions spécifiques, telles que document(), annexes() et texte_enrichi(), permettant d'aborder efficacement la complexité du projet.

Un texte enrichi par NanoML se compose d'un document et d'annexes optionnelles. Les documents et les annexes peuvent inclure un titre, du texte enrichi, des sections et des listes. Les sections fonctionnent de manière similaire aux documents et aux annexes, et peuvent être imbriquées. Les listes contiennent des items et peuvent également s'imbriquer. Le texte enrichi est une suite de mots ou de mots importants avec une mise en page spécifique.

Ensuite, la fonction document() gère la structure principale du document NanoML, en vérifiant que chaque document commence et se termine correctement avec ses balises dédiées. La fonction annexes() traite les sections additionnelles, en répétant la vérification et l'analyse du contenu dans ces blocs. La fonction texte_enrichi() analyse en combinant document() et annexes(). La fonction contenu() est une fonction qui fournit la logique de lecture et de vérification pour les éléments d'un document (sections, titres, listes) et est liée aux fonctions document(), annexe() et section() qui en dépendent.

Finalement, on a fait le choix d'utiliser des tokens, car cela permet une lecture du document plus simplifié. Les fonctions clés, telles que document(), annexes(), texte_enrichi() et contenu(),illustrent l'importance de diviser le code en plusieurs sous-parties afin de lui donner une plus grande flexibilité. L'emploi de ces tokens, identifiables via curr_tag et tag_value, garantit la cohérence de l'analyse des documents NanoML.

3. Méthodologie appliquée pour l'analyse syntaxique et la construction de l'arbre n-aire

Pour notre méthodologie d'analyse syntaxique, nous avons choisi de combiner plusieurs fonctions spécifiques, telles que document(), annexes() et texte_enrichi().

Ces fonctions nous permettent d'aborder la complexité du projet en divisant le code en plusieurs sous-parties.

Au début, nous avons analysé le document char par char, mais pour simplifier la lecture, nous avons ensuite opté pour l'utilisation de tokens comme indiqué dans la description de la grammaire.

Ces tokens sont identifiables via curr_tag et tag_value et garantissent la cohérence de l'analyse des documents.

Les fonctions clés, telles que document(), annexes(), texte_enrichi() et contenu(), utilisent ces tokens pour vérifier et analyser le contenu du document.

Les fonctions de vérification sont utilisées pour traiter ces tokens.

Suite à la vérification syntaxique par les tokens, nous utilisons une fonction principale chargée de la logique de l'arbre. Cette fonction principale est systématiquement utilisée à l'issue des opérations liées aux tokens. Les tokens, ayant pour unique rôle la vérification syntaxique, permettent, une fois cette étape validée, de procéder directement à la mise à jour de l'arbre selon les besoins identifiés.

4. Fonctionnement du programme

4.1 Description

Le programme se divise sous plusieurs fichiers. Il y a le main qui regroupe toutes les fonctions, les non-terminaux, terminaux, l'arbre et l'affichage. Les étapes pour l'utilisation et l'installation du programme sont détaillées dans le readme. Globalement, il suffit de compiler le programme et de commenter ou retirer le commentaire des fonctions que l'on souhaite utiliser dans le main.

Il y a également une addition, réalisée en python avec la fonction graphViz, qui permet de visualiser l'arbre.

Le programme utilise cette grammaire (selon la forme de Backus-Naur) :

1	<texte_enrichi> ::= <document> <annexes>
2	<document> ::= 'debut_doc' <contenu> 'fin_doc'
3	<annexes> ::= {'debut_annexe' <contenu> 'fin_annexe' }
4	<contenu> ::= { <section> | <titre> | <mot_enrichi> | <liste> }
5	<section> ::= 'debut_section' <contenu> 'fin_section'
6	<titre> ::= 'debut_titre' <texte> 'fin_titre'
7	<liste> ::= 'debut_liste' { <item> } 'fin_liste'
8	<item> ::= 'debut_item' (<liste_texte> | <texte_liste>) 'fin_item'
9	<liste_texte> ::= <liste> <texte_liste> | epsilon
10	<texte_liste> ::= <texte> <liste_texte> | epsilon
11	<texte> ::= { <mot_enrichi> }
12	<mot_enrichi> ::= <mot_simple> | <mot_important> | 'retour_a_la_ligne'
13	<mot_important> ::= 'debut_important' { <mot_simple> } 'fin_important'
14	<mot_simple> ::= { 'n'importe-quel caractère' } 'espacement'

4.2 Le main

Dans le main, il y a les variables globales utilisées par tous les autres fichiers et les appels aux fonctions gérant les différentes parties.

Parmi ces variables globales il y a le fichier, le caractère courant, le nom de la balise courante, la valeur de la balise courante, la racine de l'arbre, le nœud courant et pour finir un booléen. Ce booléen permet de spécifier simplement au programme si l'on veut utiliser ou non les fonctions liées à l'arbre. Ces variables sont définies avec le mot clé "extern" pour éviter l'instanciation de ces variables dans les autres fichiers lorsque le header est importé dans ceux-ci.

Le main gère le nom du fichier à lire (erreur si l'on ne rentre rien ou un nom invalide) et appelle toutes les fonctions "primaires" de chaque partie. On peut voir "initialiser_arbre()" qui alloue le premier nœud, mémorise la racine et spécifie que l'on souhaite utiliser l'arbre avec la variable globale citée précédemment. Ensuite il démarre la lecture du fichier et si l'on souhaite on peut sauvegarder l'arbre sous format json, l'afficher ou rendre le texte avec l'affichage "spécial". A la fin, le fichier est fermé et si l'on a utilisé l'arbre, l'arbre est libéré de la mémoire.

La structure de l'arbre est également stockée dans le header du main.

4.3 Les terminaux

Dans ce fichier, il y a les fonctions gérant les terminaux. On peut y voir les fonctions autour de la lecture des caractères, de la vérification d'un caractère, des balises, des vérifications de balises ou bien la lecture des espaces. On peut également y voir une fonction pour l'ouverture du fichier à lire.

4.4 Les non-terminaux

Dans le fichier des non-terminaux il y a les fonctions décrivant les non-terminaux de la grammaire.

C'est principalement ici que se passe la réflexion et la vérification de la syntaxe du fichier, notamment lorsqu'il s'agit de différencier entre deux non-terminaux comme "section" ou "titre".

Les fonctions s'appellent toutes "récursivement" à la façon de la POO. Nous avons essayé de faire en sorte que les fonctions ressemblent le plus possible à la grammaire pour des raisons de relecture et pour éviter tout égarement lié à un raisonnement erroné.

4.5 L'arbre

L'arbre est un arbre binaire composé d'un fils et d'un frère. Il y a également un grand-frère et un père pour remonter plus facilement dans l'arbre (afin d'ajouter un petit frère après avoir ajouté la suite de fils à un nœud, par exemple). Dans chaque nœud il y a un token de mémorisé.

Les fonctions de l'arbre sont gérées par une fonction mère "maj_arbre()". Cette fonction gère la logique pour ordonner la hiérarchie des tokens. En premier lieu elle ajoute le nœud puis effectue un traitement pour remonter le nœud courant pour que le prochain ajout se passe correctement. Par exemple, un "
" ordonne à l'arbre de remonter au premier nœud où est stocké un mot pour ajouter le prochain comme frère.

Un point **extrêmement important** est qu'il faut remonter l'arbre avant d'ajouter un token lorsqu'on ajoute une suite de mot puis un token différent d'un mot, le programme suivant une suite de mot naïvement. Pour les balises fermantes, le programme remonte tous les mots (s'il y en a) pour ensuite remonter d'un niveau dans l'arbre.

Pour l'ajout, on vérifie si le nœud possède déjà un fils, si oui on vérifie qu'il n'a pas de frère puis on l'ajoute. Dans le cas où le frère et le fils sont déjà pris, c'est qu'il y a eu

un problème au moment de remonter l'arbre. L'ajout en lui-même est un simple malloc avec le nœud courant de mis à jour par le nouveau nœud que l'on vient de créer.

Le reste des fonctions sont des fonctions pour afficher les nœuds, l'arbre et le sauvegarder. Pour ce qui est de la sauvegarde au format json, il est important de supprimer la dernière virgule avant de l'écrire dans un .json. Le fichier n'injecte pas directement le texte dans un json pour des questions d'encodage et de compatibilité.

4.6 L'affichage

D'une manière générale, l'affichage n'est qu'un vaste switch case.

L'affichage prend en considération les tabulations liées aux listes, la position de la tête d'insertion pour vérifier si un mot dépasse du cadre ou non, cadre définis par "fenêtre_actu", et la taille de la fenêtre maximale définis comme constante. Le programme mémorise également s'il faut appliquer le traitement de mot_important (commun avec titre).

L'affichage pourrait être décrit par une pseudo-grammaire. Une fenêtre étant une ligne de "+" et de "-" et une succession de barres verticales et de contenu à l'intérieur jusqu'à sa fermeture.

Les règles de cette grammaire sont dictées dans le switch case, qui lui-même est dépendant de l'arbre généré par la lecture du fichier.

Un document, un annexe ou une section ouvrent une nouvelle fenêtre, ce qui a pour effet de rétrécir la taille de l'insertion et d'ajouter une fenêtre ou une sous-fenêtre.

Chaque ligne peut être décrite comme une balise : une suite de "|" selon le nombre de fenêtres, une suite d'espaces selon les tabulations nécessaires, le texte, puis une suite d'espaces et de "|" afin de fermer la fenêtre et qu'elle soit de taille correcte puis on retourne à la ligne suivante.

Une liste décalant de 2 espaces par liste (décalages supplémentaires en cas de liste imbriquée). Les items ajoutant un "#" et décalant de 2 la tête d'insertion.

Lorsqu'il s'agit d'ajouter les mots dans cet affichage, il faut vérifier que le mot et l'espace qu'il ajoute ne dépasse pas de la fenêtre. Si le mot devrait dépasser (selon la formule "position de la tête d'insertion + taille du mot + un espace"), alors on finalise la fenêtre ce qui a pour effet d'ajouter des espaces pour combler, puis on ferme la bordure de la fenêtre avant de retourner à la ligne suivante, où on y "ouvre" la ligne avant d'ajouter le mot. Dans le cas où tout se passe bien, alors on ajoute juste le mot. Dans les deux cas, on met le mot en majuscules s'il doit être important (selon le booléen cité plus haut) et on ignore les mots vides.

4.7 La visualisation de l'arbre

La visualisation de l'arbre se fait dans un autre dossier. Dans ce dossier, on y retrouve un dossier contenant les fichiers source, un autre dossier contenant les images et les json à lire.

Le fichier Arbre.py est une classe qui contient un constructeur initialisant une liste vide. Cette liste vide est remplie grâce à la méthode "charger()" qui lit le json.

Le json contient tous les nœuds de l'arbre, chaque nœud est décrit par son adresse, l'identifiant du token, la valeur du token et les adresses de son fils, petit frère, grand frère et père.

Une fois le json lu, la méthode "to_png()" se charge de compiler toutes les données. La méthode utilise la bibliothèque graphViz.

Il suffit de changer le nom du fichier dans le main.py pour visualiser un autre arbre. Sous condition que le fichier .json existe.

5. Discussion sur la mise en œuvre de la visualisation de l'arbre et les défis rencontrés.

L'une des difficultés rencontrées au début de notre projet a été l'utilisation de l'analyse char par char. Cette approche s'est avérée peu efficace et difficile à tester. Nous avons donc opté pour l'utilisation de tokens suite aux conseils et réponses données lors de la Faq.

L'utilisation de tokens nous a permis de simplifier les tests et de gérer plus facilement les cas où il est nécessaire de reprendre l'étiquette de la balise précédente ou suivante.

Cela nous a également permis de savoir quand continuer, arrêter une boucle ou une condition, ou renvoyer une erreur.

Un autre défi rencontré a été la définition de la structure de l'arbre.

Il a été difficile de trouver la bonne structure pour représenter correctement la hiérarchie des éléments du document.

Un autre défi technique a été l'intégration des balises orphelines, telles que
, qui ne s'inscrivent pas dans la hiérarchie classique des éléments parents et enfants. Leur traitement spécifique a requis des ajustements dans notre logique de construction afin d'assurer leur placement correct sans perturber la structure globale.

De plus, la distinction entre une simple suite de mots et une suite nécessitant un remontage dans la hiérarchie de l'arbre pour trouver le bon parent a représenté un casse-tête supplémentaire. Cette difficulté résidait dans la création d'une logique capable d'interpréter correctement quand et comment modifier la structure de l'arbre pour refléter fidèlement la structure du document, sans introduire d'erreurs dans la représentation des relations entre les éléments.

Finalement, la visualisation de l'arbre est également contraignante, car il était difficile de déboguer et de comprendre comment la structure était construite.

Pour surmonter certaines de ces difficultés, nous avons utilisé un projet de théorie des langages pour déboguer et visualiser la structure de l'arbre.

Cela nous a permis de comprendre comment la structure était construite et de déboguer plus facilement les erreurs.

6. Analyse des résultats obtenus et des améliorations potentielles.

Dans la première phase du projet, nous nous sommes concentrés sur l'analyse syntaxique et la validation de la structure du document selon la grammaire NanoML. Cependant, cette approche présentait des limitations importantes car nous ne construisions pas de représentation explicite du document en mémoire. Nous nous contentions de lire, valider et interpréter les balises ainsi que les fichiers texte.

L'une des difficultés rencontrées a été l'utilisation de l'analyse char par char, qui s'est avérée peu efficace et difficile à tester. Nous avons donc opté pour l'utilisation de tokens, ce qui nous a permis de simplifier les tests et de gérer plus facilement les cas où il est nécessaire de reprendre l'étiquette de la balise précédente ou suivante.

Dans la deuxième phase du projet, nous avons introduit une représentation sous forme d’arbre reflétant la structure hiérarchique et les relations parent-enfant entre les éléments du document. Il est important de souligner que l'arbre ne sert qu'à ordonner les balises pour préparer l'affichage ou tout autre type de traitement. Il aurait été difficile de réaliser un traitement supplémentaire, comme l'affichage, sans la mise en place de cette représentation sous forme d'arbre.

Par la suite, l'adaptation des fichiers terminaux.c et non_terminaux.c pour intégrer la logique de l'arbre a été décisive dans l'analyse syntaxique. Désormais, la lecture d'une balise ou la modification d'une valeur entraîne une mise à jour correspondante dans l'arbre. Pourtant, bien que l'arbre fonctionne pour la grammaire spécifiée, il copie naïvement tous les tokens.

En effet, la structure de l'arbre est rigide et sensible à la casse, et elle copie naïvement tous les tokens. Des améliorations potentielles pourraient inclure l'utilisation d'un type de structure ou d'un paradigme différent pour donner une seconde couche de traitement à l'arbre et le rendre plus flexible.

Pour déboguer le projet et améliorer la visualisation, Charles a eu l'idée d'utiliser le projet de théorie des langages.

Ce projet offre une représentation visuelle, ce qui est idéal pour visualiser les relations parent-enfant et déboguer efficacement.

Cette approche nous a permis de représenter visuellement notre "DOM" NanoML sous forme d'arborescence.

En ce qui concerne les potentielles améliorations de ce projet, nous pouvons facilement envisager l'ajout de différents tokens pour lire la grammaire et interpréter différentes balises autre que .

Il serait également plus intéressant d’utiliser la récursivité plutôt que les while dans les non-terminaux. Les whiles forçant le programme à tout lire/exécuter tandis que la récursivité nous permettrait de “diviser” le programme et de mieux gérer quand est-ce que la boucle devrait recommencer.

Si nous voulons aller plus loin et être fidèle au "DOM" de l'HTML, nous pourrions créer différents tokens pour simuler des "attributs" et ainsi avoir différents comportements à la lecture de ces derniers

Cela pourrait ressembler à cela :

"<element> ::= ‘debut_element’ {<attribut>} <contenu> ‘fin_element’, où {<attribut>} représente une répétition d'attributs potentiels pour cet élément."

<attribut> ::= ‘nom_attribut’ "valeur_attribut", où nom_attribut pourrait être "src", "href", etc., et "valeur_attribut" la valeur associée à cet attribut.

Une autre idée d'amélioration serait de rendre la grammaire modifiable et stockable dans un fichier, et d'agencer les fonctions en fonction de la grammaire donnée. Ainsi que de pouvoir sauvegarder l'arbre, le rendre dynamique, et créer une interface sous forme d'app.

Finalement, l’affichage est assez direct. Il suit l’arbre récursivement et applique les règles prédéfinies pour chaque nœud. L’affichage étant presque complètement dépendant des précédents éléments, les améliorations sont assez limitées. L’affichage se faisant dans la console, cela limite également encore plus les possibilités.

7. Conclusion générale et perspectives d'évolution du projet.

Dans le cadre de notre projet NanoML, nous avons compris comment une grammaire peut servir à vérifier la conformité syntaxique d'un document et à quel point l'utilisation d'enum peut simplifier la gestion des différents éléments d'un document. (spécialement dans :). Puis nous avons découvert les cas d'usage pratiques d'un arbre avec la représentation de la structure hiérarchique d'un document.

De plus, en utilisant une approche divisée en plusieurs fonctions spécifiques, nous avons pu analyser la complexité du projet et simplifier la lecture des documents en utilisant des tokens. Les fonctions clés telles que document(), annexes(), texte_enrichi()

et contenu() ont permis de vérifier et d'analyser le contenu du document en utilisant ces tokens.

Ensuite, nous avons rencontré plusieurs défis dans la mise en œuvre de la visualisation de l'arbre, notamment l'intégration des balises orphelines et la distinction entre une simple suite de mots et une suite nécessitant un remontage dans la hiérarchie de l'arbre. Cependant, nous avons pu surmonter ces difficultés en utilisant un projet de théorie des langages pour déboguer et visualiser la structure de l'arbre.

Notre projet NanoML nous a permis de comprendre l'importance de la vérification syntaxique et de la représentation sous forme d'arbre dans la gestion de documents structurés. Nous avons également découvert comment l'utilisation d'enum aurait pu simplifier la gestion des différents éléments d'un document.

Cependant, pour une éventuelle suite du projet, nous pouvons envisager d'apporter des améliorations supplémentaires à notre projet, notamment en utilisant une structure ou un paradigme différent pour rendre l'arbre plus flexible et en ajoutant des fonctionnalités d'affichage et d'interface utilisateur.